09/13/00 JC932 U.S. PTO

09-15-00

A

JC675 U.S. PTO 09/663483 09/13/00

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of: Abraham R. Matthews et al.

Title: SWITCH MANAGEMENT SYSTEM AND METHOD

Attorney Docket No.: 1384.006US1

PATENT APPLICATION TRANSMITTAL

BOX PATENT APPLICATION
Commissioner for Patents
Washington, D.C. 20231

We are transmitting herewith the following attached items and information (as indicated with an "X"):

X Return postcard.
X Utility Patent Application under 37 CFR § 1.53(b) comprising:
- X Specification (31 pgs, including claims numbered 1 through 3 and a 1 page Abstract).
- X Formal Drawing(s) (29 sheets).
- X Unsigned Combined Declaration and Power of Attorney (3 pgs).
- X Appendix A (9 pgs.)
- X Appendix B (5 pgs.)

The filing fee (NOT ENCLOSED) will be calculated as follows:

	No. Filed	No. Extra	Rate	Fee
TOTAL CLAIMS	3 - 20 =	0	x 18 =	$0.00
INDEPENDENT CLAIMS	3 - 3 =	0	x 78 =	$0.00
[ ] MULTIPLE DEPENDENT CLAIMS PRESENTED				$0.00
BASIC FEE				$690.00
TOTAL				$690.00

THE FILING FEE WILL BE PAID UPON RECEIPT OF THE NOTICE TO FILE MISSING PARTS.

SCHWEGMAN, LUNDBERG, WOESSNER & KLUTH, P.A.
P.O. Box 2938, Minneapolis, MN 55402 (612-373-6900)

By: Thomas F. Brennan
Atty: Thomas F. Brennan
Reg. No. 35,075

Customer Number 21186

"Express Mail" mailing label number: EL618477208US
Date of Deposit: September 13, 2000
This paper or fee is being deposited on the date indicated above with the United States Postal Service pursuant to 37 CFR 1.10, and is addressed to the Commissioner for Patents, Box Patent Application, Washington, D.C. 20231.

UNITED STATES PATENT APPLICATION

SWITCH MANAGEMENT SYSTEM AND METHOD

INVENTORS

Abraham Rabindranath Matthews

Anna Berenberg

Schwegman, Lundberg, Woessner, & Kluth, P.A.
1600 TCF Tower
121 South Eighth Street
Minneapolis, Minnesota 55402
ATTORNEY DOCKET 1384.006US1

SWITCH MANAGEMENT SYSTEM AND METHOD

Field of the Invention

The present invention is related to networking systems, and more particularly to a system and method for managing a switch within a wide area network (WAN).

Background Information

Internet or WAN service providers are operating in a crowded marketplace where cost effectiveness is critical. Operational costs present a significant challenge to service providers. Cumbersome, manual provisioning processes are the primary culprits. Customer orders must be manually entered and processed through numerous antiquated back-end systems that have been pieced together. Once the order has been processed, a truck roll is required for onsite installation and configuration of Customer Premises Equipment (CPE), as well as subsequent troubleshooting tasks.

Presently, the delivery of firewall services requires the deployment of a specialized piece of Customer Premises Equipment (CPE) to every network to be protected. This model of service delivery creates an expensive up-front capital investment, as well as significant operational expenses that are associated with onsite installation and management of thousands of distributed devices. The results are service delivery delays, increased customer start-up costs and/or thinner service provider margins.

The slow and expensive process of deploying firewall services cuts into margins and forces significant up-front charges to be imposed on the customer. In order to be successful in today's market, service providers must leverage the public network to offer high-value, differentiated services that maximize margins while controlling capital and operational costs. These services must be rapidly provisioned and centrally managed so that time-to-market and, more importantly, time-to-revenue are minimized. Traditional methods of data network service creation, deployment, and management present significant challenges to accomplishing these goals, calling for a new network service model to be implemented.

Enterprise customers are increasingly demanding cost-effective, outsourced connectivity and security services, such as Virtual Private Networks (VPNs) and managed firewall services. Enterprise networks are no longer segregated from the outside world; IT managers are facing mounting pressure to connect disparate business units, satellite sites, business partners, and suppliers to their corporate network, and then to the Internet. This raises a multitude of security concerns that are often beyond the core competencies of enterprise IT departments. To compound the problem, skilled IT talent is an extremely scarce resource. Service providers, with expert staff and world-class technology and facilities, are well positioned to deliver these services to enterprise customers.

What is needed is a system and method for providing managed network services that are customizable for each customer's need. Furthermore, what is needed is a system and method for controlling such managed network services.

Summary of the Invention

According to one aspect of the present invention, a system and method of managing a switch includes installing a switch having a plurality of processor elements, installing an operating system on each processor element, creating a system virtual router and configuring the processor elements from the system virtual router.

According to another aspect of the present invention, a switch management system includes an object manager and a distributed management layer, wherein the object manager communicates with objects through the distributed management layer.

Description of the Preferred Embodiments

In the following detailed description of the preferred embodiments, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the ways used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar computing device, that manipulates and transforms data represented as physical (e.g., electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

While IT managers clearly see the value in utilizing managed network services, there are still barriers to adoption. Perhaps the most significant of these is the fear of losing control of the network to the service provider. In order to ease this fear, a successful managed network service offering must provide comprehensive visibility to the customer, enabling them to view configurations and performance statistics, as well as to request updates and changes. By providing IT managers with powerful Customer Network Management (CNM) tools, one can bolsters confidence in the managed network service provider and can actually streamline the service provisioning and maintenance cycle.

While service providers recognize the tremendous revenue potential of managed firewall services, the cost of deploying, managing and maintaining such services via traditional CPE-based methods is somewhat daunting. Service providers are now seeking new service delivery mechanisms that minimize capital and operational costs while enabling high-margin, value-added public network services that are easily provisioned, managed, and repeated. Rolling out a network-based managed firewall service is a promising means by which to accomplish this. Deploying an IP Service Delivery Platform in the service provider network brings the intelligence of a managed firewall service out of the customer premises and into the service provider's realm of control.

One such IP Service Delivery Platform 10 is shown in Fig. 1. In the embodiment shown in Fig. 1, IP Service Delivery Platform 10 includes three distinct components: an intelligent, highly scalable IP Service Processing Switch 12, a comprehensive Service Management System (SMS) 14 and a powerful Customer Network Management (CNM) system 16. Service Management System (SMS) 14 is used to enable rapid service provisioning and centralized system management. Customer Network Management (CNM) system 16 provides enterprise customers with detailed network and service performance systems, enable self-provisioning. At the same time, system 16 eases IT managers fears of losing control of managed network services.

In one embodiment, such as is shown in Fig. 2 for a network-based managed firewall service model, the service provider replaces the high-capacity access concentration router at the POP with an IP Service Processing Switch 12. This is a higher-capacity, more robust, and more intelligent access switch, with scalable processing up to 100+ RISC CPUs. Just as with the access router, additional customer access capacity is added via installing additional port access blades to the IP Service Processing Switch chassis. Unlike conventional access routers, however, additional processor blades can be added to switch 12 to ensure wire-speed performance and service processing.

The intelligence resident in IP Service Processing Switch 12 eliminates the need to deploy CPE devices at each protected customer site. Deployment, configuration, and management of the managed firewall service all take place between IP Service Processing Switch 12 and its Service Management System 14. In the embodiment shown in Fig. 2, Service Management System 14 resides on a high-end UNIX platform at the service provider NOC.

In one embodiment, the customer has the ability to initiate service provisioning and augmentation via a web-based Customer Network Management tool residing, e.g., at the customer's headquarters site. This is an entirely different service delivery paradigm, requiring little or no truck rolls and little or no on-site intervention.

In one embodiment, switch 12 is a 26-slot services processing switch that marries scalable switching, routing and computing resources with an open software architecture to deliver computationally-intense IP services such as VPNs with scalable high performance. In one embodiment, switch 12 has a high-speed 22 Gbps redundant dual counter-rotating ring midplane. Slots are configured with four types of Service Blades: Control, Access, Trunk and Processor blades with specialized processing which enables a range of high-performance services including route forwarding, encryption and firewalls.

Service providers can use switch 12's virtual routing capabilities, and its ability to turn IP services into discrete and customized objects, to segment and layer services for the first time for tens of thousands of discrete subscriber corporations. In addition, processor capacity can be added to switch 12 by adding new processor blades.

In one embodiment switch 12 includes an operating system which dynamically distributes services to switch 12 processors.

In one embodiment, the 26-slot services processing switch corrects for failures using the redundant counter-rotating ring midplane.

In one embodiment, each Service Blade automatically fails-over to a backup.

One embodiment of a switch 12 is described in "System and Apparatus for Delivering Security Services," filed herewith, the description is incorporated herein by reference.

In one embodiment, switch 12 is designed to integrate seamlessly into a SP's preexisting network, whether that be through support of open routing protocols or through its Frame Relay to IPSec interworking solution that integrates new IP-based networks into a corporation's preexisting Frame Relay cloud.

The operating system will be described next.

In one embodiment, switch 12 includes a network operating system (NOS) 20. In one embodiment, network operating system 20 enables switch 12 to create discrete customized services to specific subscriber corporations by providing them each with a different configuration of service object groups. NOS 20 enables objects within these object groups to be distributed dynamically to customized processors so that application services are receiving the right level of computational support.

In one embodiment, NOS 20 is designed as an open Application Program Interface (API) that allows general-purpose software or new advanced IP services to be ported into the platform from best of breed third parties in a continual fashion, helping to enrich service provider' investment over time.

In one embodiment, NOS 20 includes a distributed messaging layer (DML) 22 component, an object manager (OM) component 24 layered over DML, control blade redundancy (CBR) 26 for redundant system controllers, a resource manager (RM) 28 for managing separate resource elements and a resource location service (RLS) 30. Resource location service 30 provides load balancing across capable processor elements (Pes) to create an object. PE selection is based on predefined constraints.

In one embodiment, CBR 26 is layered over DML 22 and OM 24.

In one embodiment, Object Manager 24 consists of three layers as shown on Fig. 4. The upper layer titled *OM Controller and Database* (OMCD) 40 is concerned with managing the VPN and VR configuration. This is the agent that deals with the configuration manager directly. Middle layer 42 entitled *OM Object Routing and Interface Global* is concerned with managing global (across the switch system) object groups and object configurations. Lower layer 44 entitled *OM Object Routing and Interface* (OMORI) is concerned with managing local objects and groups as well as routing control information between address spaces based on the location of objects, and interfacing with the object via method invocation.

In one embodiment, the IPSX object database consists of two types of databases: Global (managed on Master Control Blade by OMORIG) and distributed local databases (managed by OMORI agents on every PE present in the system). In one such embodiment, the global database is a superset of the extracts from local databases.

Objects represent a basic unit of management for purposes of fault tolerance, computational load balancing, etc. One or more adjacent protocol modules can be placed into a single object. It is also possible that a module is split across two objects.

In IP, each host has a globally unique IP Address. Additionally each type of transport on top of IP has a globally unique Protocol ID. Each application on top of a transport has a Local Port Number that is unique locally. Thus an application instance in the network is uniquely identified by the tuple

<IP Address, Protocol ID, Port Number>

In switch 12, each Processing Element (PE) has a globally unique PEID. Each type of object in a PE has a globally unique Type ID. Within each type, objects are assigned locally unique numbers or ID. Thus within a switch 12, each object is uniquely identified (analogous to IP applications) by

<PEID, Object Type, Local Object ID>

The format of the Local Object ID is dependent on the object type. Mostly, driver and IO Layer objects have IDs that are constructed based on physical attributes. The physical attributes used are

blade	A globally unique Blade ID (e.g. slot number).
port	A locally unique Port ID for all ports within a blade.
channel	A locally unique Channel ID for all channels within a port. (This may be dynamically variable as in Channellized DS3.)
vcid	A locally unique ID that is assigned by the Link Layer protocol agent, e.g., is a FR DLCI.

The following function is used to initialize an Object ID for any object created by the Object Manager. (e.g. object id type is OBJ_ID_TYPE_OBJECT).

```
void obj_id_init (
      object_id_t                 *id,
      object_type_t               type,
      object_group_id_t           group,
      local_object_id_t           object);
```

The following function is used to initialize an Object ID for any object created by the IO Layer. IO Layer objects are created either at system startup time by the device driver sub-system or by a Link Layer protocol module in response to an IOCTL. The important thing to note is that it is not created by Object Manager 24.

```
void obj_id_init_physical (
      object_id_t                    *id,
      object_type_t                  type,
      physical_id_t                  blade,
      physical_id_t                  port,
      physical_id_t                  channel,
      physical_id_t                  vcid);
```

Group is an aggregation point for all objects that comprises the VR. Group and VR have one-to-one mapping. A Group encompasses objects, which are located in different address spaces. Group Id, which identifies a group, is unique in the scope of a single switch 12.

Figure 5 shows the distinction between an Object Class and an Object Group. Both are collections of objects. As shown in Figure 5, an object class is a set of objects that have the same type signature and behavior (e.g. Applications Class 50, TCP/IP Class 52 and Interfaces Class 54). In contrast, for an object group, the constituent objects do not necessarily have the same type signature and behavior (e.g. Object Groups 1 to 3). There can be multiple objects of the same class in an object group (e.g. Object Group 2 has two objects of Interface Class). On the other hand, an object group need not have an object of each class (e.g. Object Group 3 does not have an object of Interface Class).

In one embodiment, OMCD 40 is the agent, which interfaces to the Configuration Manager. As shown on Figure 6 OMCD 40 manages 1) the Global list of VPN in system 10; and 2) the list of VRs per VPN. The caveats for VPN and VRs are:

- VPN ID equal to 0 is illegal;
- Global uniqueness of VPN ID across IPSX systems is the responsibility of the Service Management System (SMS).

In one embodiment, OMCD 40 creates a vpn descriptor every time Configuration managers request VPN creation. Every VPN is identified by a unique VPN ID. In one embodiment, each Virtual Router (VR) is identified by a VR ID, which is the IP Address of the VR. VR ID is unique in the VPN context. When Configuration Manager requests creation of an existing VR in the VPN, VR creation request is rejected. Otherwise a vr descriptor will be created.

There are several types of the VR:

1) ISP (Internet Service Provider) VR: Typically there is 1 such VR for a single switch 10.
2) Control VR: There can be only one Control VR for a single switch 10. This VR is used to host the management applications such as SNMP, Telnet etc.
3) Customer VR: There are several Customer VRs in a single switch 10. Typically, there is one Customer VR per customer service point.

Detailed VR creation process is described below.

OMORIG agent 42 runs on every Control Blade, whether it is Master or Standby Blade. OMORI local sends the change only to Master. Control Blade Redundancy feature, described below**Error! Reference source not found.** takes care of replicating and synchronizing OMORIG database from Master to Standby.

OMORIG 42 provides several mappings of Object Ids. It manages lists of object Ids, which are located on the same address space, lists of object Ids which belong to the same group, a sorted Global object ID list and an unsorted Global object ID list. The OID link is shown on the Figure 7. The OID link data structure and OMORIG API are described in Appendix A.

OMORI is the OM agent. OMORI runs on every processing node and manages local objects and forwards IOCTLs to another object, whether local or remote. OMORI for each object creates `object_descriptor_t`, which has all the management information.

As shown on Figure 8, OMORI manages a global list of local object descriptors 60, a global list of groups 62, and a list of object descriptors per group 64.

Each change in the OMORI database is propagated to the OMORIG, which runs on the Active Master. OMORI sends separate messages, varying by message tag, per each action to be taken to change Global Database.

OMORI additionally serves the request from the object on information about any other object. If requested object local to OMORI then it finds all the data in the local database otherwise OMORI agent forwards such a request to OMORIG, which has validated data.

The creation and deletion of objects in an object group needs to be coordinated. The issues to be dealt with are as follows. First, an object may need to IOCTL another object for correct setup or shutdown. We need to ensure that all default objects for the group are present.

Second, an object when using a cached pointer must be sure that it has not become stale.

Every OMORI maintains a state machine for each local object. Each object is supposed to take an appropriate action on every state change notification from the OMORI. Object State Transition Diagram is in the Figure 9 and detailed description is on the Table 1.

The caveats for the object states are as follows. First, in *init* state, the object's base data structure is created and initialized. Essential auxiliary data structures may also be created and initialized.

Second, in *stopped* state, no IOCTL or data send can take place. All non-essential auxiliary data structures must be deallocated. Only the object's base and essential data structures may remain allocated. All cached pointers should be released. All system resources (e.g. timers) must be deactivated. The object may be the target of IOCTLs and is expected to respond gracefully. The object should never initiate an IOCTL – either directly or in response to another IOCTL

Third, in *active* state, non-essential auxiliary data structures and system resources are activated. The object may cache pointers. The object can initiate and respond to IOCTLs.

Fourth, in *dead* state, all (object's base and essential auxiliary) data structures are deallocated.

STATE	EVENT	ACTION
INIT	Receive Create Object Request	Cal constructor for the object class. Transit to STOPPED
STOPPED	RECV ACTIVATE request	Send ACTIVATE_OBJECT generic IOCTL, if completed with SUCCESS transit to ACTIVE.
STOPPED	RECV DESTROY request	Send DESTROY_OBJECT generic IOCTL, transit to DEAD, remove object descriptor from list, free it.
ACTIVE	RECV DESTROY request	Transit to STOPPED state. Send DESTROY_OBJECT generic IOCTL, transit to DEAD, remove object descriptor from list, free it.
ACTIVE	RECV ACTIVATE request	
STOPPED	RECV ADD to Group /REMOVE from Group request	Modifies group membership as requested
ACTIVE	RECV ADD to Group /REMOVE from Group request	Transit to STOPPED state. Modifies group membership as requested

Table 1. Object State Machine

Distributed Messaging Layer (DML) 22 is used to provide inter-processor communication and isolated channels for data and control messages as is shown in Figure 10.

OMORIG and OMORI communicate via predefined DML channel DML_CHAN_DATA. All IPNOS nodes in the system are members of DML_CHAN_DATA. During initialization process OMORI register to DML receive function, which will be called every time a new packet arrives on DML_CHAN_DATA. OMORIG and OMORI are DML applications and therefore they are notified on every dynamic event in the system.

There are four types of dynamic events indicated by DML. These are:

- Peer Up – new IPNOS node detected and reachable.
- Peer Down – existing IPNOS node became unreachable
- Master Up – new Master elected in the system
- Master Down – existing Master became unreachable

On peer down event OMORI agent aborts all the pending transactions associated with the peer that went down. In addition, on a peer down event OMORIG destroys in its database all the objects that are local to the peer that went down. After that a scrub of the database is done. This includes destroying all groups which do not have any objects in them and destroying any VR associated with that group.

On peer up event and master up event OMORIG agent runs global database update protocol shown in Figure 11. In addition, on peer up event OMORIG agent initiates a database update for local objects of the new peer. OMORIG maintains state machine per OMORI. Global Database Update State Transition Diagram is shown in Figure 11. A detailed description of the transitions is in Table 2.

STATE	EVENT	ACTION
START	TIMEOUT && (request count < MAX)	Send update request
START	TIMEOUT && (request count > MAX)	Peer did not reply. Update FAILED Transit to FINISH state.
START	RECV UPDATE GROUP message	Transit to UPDATE GROUP state. Set last update equal to the current time.
START	RECV UPDATE OBJECT message	Transit to UPDATE OBJECT state. Set last update equal to the current time.
UPDATE GROUP	RECV UPDATE GROUP message	Set last update equal to the current time. Modify Database
UPDATE GROUP	RECV UPDATE GROUP DONE message	Transit to UPDATE OBJECT state. Set last update equal to the current time.
UPDATE GROUP	TIMEOUT && (delay > MAX)	Transit to FINISH state.
UPDATE OBJECT	TIMEOUT && (delay > MAX)	Transit to FINISH state.

UPDATE OBJECT	RECV UPDATE OBJECT DONE message	Transit to FINISH state.
UPDATE OBJECT	RECV UPDATE OBJECT message	Set last update equal to the current time. Modify Database

Table 2. Global Database Update Protocol

The same protocol is initiated on a master up event by the OMORIG agent to all peers that are known at that moment.

As described above, when peer goes down all virtual routers (VRs) as well as groups and objects, associated with that peer, are removed from the Global Database. If for some reason switch 12 becomes partitioned and then coalesces back, the problem with dangling objects arises, because all the objects still exists on the isolated peer. To address this a database reconciliation process is provided.

On a peer up event, the global database update protocol is started. When an update group message is received and the group is not found in the group list then

1) Look up VR by VPN ID and VR ID from the update message. If not found, then need to recreate dependencies by the following algorithm:

- Check whether VPN with VPN ID, from the update message, exists. If not then create VPN with the specified VPN ID.
- Check whether VR with VR ID, from the update message, exists. If not then create VR with the specified VR ID.
- Create group with ID received from the update message.

2) VPN/VR found: Send message to the OMORI, which send an update message to remove, specified group.

Transaction Layer 46 in Figure 10 is used to provide management of request/ reply transaction and has *exactly once* semantics, with guaranteed termination. There is a global list of requests on each processor. Each request is identified by unique index. When request is being sent a callback function could be provided if reply is expected. One embodiment of Transaction Layer Data structures and an associated API are described in Appendix A.

Object creation and communication will be described next.

In Figure 12, the communication for object creation is shown. IP object with OID 1 requests Firewall object to be created. OM creates object descriptor and based on the specified class of the object (e.g. Firewall), OM finds the appropriate *constructor* function in the object class table and constructs the new object. The same algorithm is used for management communications between objects (IOCTL). Based on the class id of the destination object appropriate *control* function from the object class table is called. It is the responsibility of the object implementers is to supply handlers for all supported IOCTLs.

The IOCTL mechanism is described below. Typically IOCTL between objects is used for Management Plane communications. For Data Plane communications between objects, object to object channels are used.

Objects can be created in three different ways:

- **REGISTERED**: Created as system comes up (e.g. drivers) and registered to the OMORI with object id, having physical location meaning.
- **CREATED BY OM**: Created by Object Manager. In this case OMORI creates a locally unique (in the scope of this address space) object ID that in conjunction with address space id gives unique object id inside of the system. To create an object, the constructor function based on the object class will be called.
- **ASSIGNED and REGISTERED**: This is hybrid of two cases described above. Object created without OM, but requests OM to assign unique Object ID to it and registers with this ID to OMORI. Typically this is used by the Resource Manager.

OM 24 issues control codes when invoking the control method for an object. To ease the management of these control codes, a module based code scheme is used.

Every code is composed of an implicit module identifier and a module specific code identifier. The macro *OBJ_CTL_CODE(module, module_specific_code)* is used to construct the object control code.

```
#define OBJ_CTL_CODE(m,c)      (((m) << 24) | (c))
#define OBJ_CTL_MODULE(c)      ((c) >> 24)
```

Generic IOCTL are primarily, used by Object Manager 24, to inform all the objects in the specified group of some state change. For example, when user requests to delete VR, before removing all the objects in the VR's group, STOP_OBJECT generic IOCTL is sent to every object in the group. MODULE_ALL is used as a module identifier for all the generic IOCTLs.

Every object should support the following generic IOCTLs:

ACTIVATE_OBJECT
STOP_OBJECT
DESTROY_OBJECT

OM 24 does not interpret the code component. The object shell breaks the control code in to a module identifier and a module specific code. It then issues the module specific code to a module specific control function.

Objects can be destroyed in two different ways:

DEREGISTERED: Objects which were registered on creation will be deregistered on destruction

DESTROYED: There is no explicit destructor to destroy an object, instead generic IOCTL DESTROY_OBJECT is send to object, which is to be destroyed.

The IOCTL mechanism provides a reliable transaction oriented inter-object communication that is suitable for management and control traffic. However, the IOCTL based communication is not fast or efficient. For protocol data traffic, a lighter, faster and efficient mechanism is needed.

In one embodiment, object channels provide a *Point-to-point* (P-P) communication mechanism that is based on a *send-and-forget* model of programming. Packets arriving at an object channel are delivered to the object asynchronously.

Objects maintain a list of *Connection End Points* (CEP). Each CEP is assigned an index that is unique within the scope of the object. Since the object's OID is globally unique, a globally unique CEP-ID is generated by the tuple

<OID, Type, index>

The *type* parameter is used to distinguish between different classes of CEPs. (E.g. the IP Forwarding object has CEPs for Virtual Interfaces and CEPs to cryptographic resources.) The CEP is represented in IPNOS by the *obj_comm_t* data structure.

Each object allocates a CEP (which is typically embedded within an object specific structure). The object then initializes the CEP by the function

```
extern  int      obj_init_channel (
        obj_comm_t            *channel,        /* Channel to init */
        void                  *object,/* Object channel is associated with */
        obj_comm_service_f    *service         /* Rx Packet service handler */
        );
```

The *service* parameter for *obj_init_channel* is an upcall handler function that is called when a packet is received on the channel. The *object* parameter is passed as the first parameter of the upcall handler, and is typically used to identify the data structure that the channel is embedded in.

After a CEP has been initialized, it can be connected to another CEP via

```
int     obj_associate_channel (
        obj_comm_t          *local_chan,    /* Local channel */
        obj_cep_id_t   *local_cep,          /* Local channel ID */
        obj_cep_id_t   *remote_cep          /* Remote channel ID */
        );
```

A CEP that is connected can be disconnected via

```
int obj_disassociate_channel (
        obj_comm_t          *local_chan             /* Local channel  */
        );
```

Sometimes it is necessary that a CEP be loopbacked to itself. This can be done by

```
int  obj_loopback_channel (
     obj_comm_t          *local_chan,    /* Local channel  */
     obj_cep_id_t   *local_cep           /* Local channel ID */
     );
```

Figure 13 shows the linkages when two CEPs within the same address space are connected. Note that

1. The CEP has a *send* and a *receive* section (of type *obj_comm_params_t*).
2. The *send* section of Obj-1 is updated with the contents of the *receive* section of Obj-2.
3. The *send* section of Obj-2 is updated with the contents of the *receive* section of Obj-1.
4. The *Remote CEP-ID* of each object (of type *obj_cep_id_t*) contains the CEP-ID of the CEP at the other end.
5. The field *remote_srv_cb* points to the remote CEP. (This is not used when the CEPs are in different address spaces.

When Obj-1 sends a packet to Obj-2, it becomes a function call. The overhead is very little. The sequence of actions that takes place when `obj_associate_channel` is called is shown in Table 3.

Step	Local CEP Object	IPNOS	Remote CEP Object
1	`obj_associate_ channel(local_ chan,`		

	local_cep_id, remote_cep_id)		
2		omori_obj_ioctl_by_id (&remote->object, remote_cep_id->object.group, OBJ_CTL_CODE (remote_cep_id ->module_id, GET_CEP_ADDR), &cep, sizeof (get_cep_addr_t) Remote_chan = cep.address	
3			In GET_CEP_ADDR IOCTL handler, return CEP's address.
4		Copy CEP IDs to remote and local	
5		Setup local channel pointers	
6		Setup remote channel pointers	

Table 3: Connecting local CEPs

Figures 14 and 15 show how services can be *pushed* onto a channel that has been established. (Note that the services and parameters should be *pushed* after a connection has been established. Services and service parameters from an earlier connection can not be assumed to be in effect.)

To enable a service, use the function

```
int     obj_update_service_params_on_channel (
        obj_comm_t                  *channel,
        int                         service_id,
        int                         direction,
        int                         operation,
        void                *params
        );
```

To disable a service, use the function

```
int     obj_disable_service_on_channel (
        obj_comm_t              *channel,
        int                     service_id,
        int                     direction
        );
```

To update the parameters for a service, use the function

```
int     obj_update_service_params_on_channel (
        obj_comm_t              *channel,
        int                     service_id,
        int                     direction,
        int                     operation,
        void              *params
        );
```

Note that in Figure 14, only the local CEP is modified when a service is enabled on transmit. In Figure 15 on the other hand, the remote CEP is modified when a service is enabled on receive.

The services that are currently supported are:

OBJ_COMM_SRV_NONE	This is never used. It is used to indicate the CEP base.
OBJ_COMM_SRV_UNSPECIFIED	This is never used. May be used to indicate errors.
OBJ_COMM_SRV_REMOTE	This service provides transport between PEs (aka address spaces). This service is automatically pushed, by IPNOS, on both receive and transmit at both ends of the channel, when the CEPs are in different PEs.
OBJ_COMM_SRV_LOCALQ	In a VI-VI connection, this is used to breakup the function call chain. It is used only when the VI CEPs are both in a single PE.
OBJ_COMM_SRV_RAQ	This is used to enforce a specific rate. The integration interval used is that of a single OS Clock Tick. It is currently not used. In the future a better algorithm based on the *Leaky Bucket* should be used.

Connecting CEPs in different address spaces (aka PEs) is more complex. IPNOS uses channel services to bridge the address spaces. The specific service that is used is *OBJ_COMM_SRV_REMOTE*. The steps taken by NOS 20 are shown in Figure 16.

Connecting remote CEPs involves the two objects, NOS 20 on both PEs, Resource Manager and Logical Queue Manager on both PEs. Figure 17 shows the configuration when remote CEPs are connected.

A method of using OM 24 will be described next in the context of VR creation. To explain VR creation process a simplified version of a single object in the group is used.

As shown on the Figure 18, user requests creation of VR 1.1.1.1 for VPN 1 on the blade with id 2. (This implies that VPN 1 was created prior to the described request.) In one embodiment, the steps described in Figure 19 will be taken.

As shown on Figure 20, user requests to create VR 1.1.1.1 for VPN 1 on blade with id 2. This implies that VPN 1 was created prior to the described request. VR consists of multiple objects. As an example here IP object, trace route object (TR), and SNMP object encompass VR. In one embodiment, the steps described in Figure 21 will be taken.

A complement operation to create VR with multiple objects in the group is to destroy such a VR. Destroy VR operation is shown on the Figure 22. In one embodiment, the sequence of steps taken is shown in Figure 23.

Scalability issues will be discussed next. An IOCTL to the object, which is located on the processor other than originator of the IOCTL, causes IOCTL to be forwarded to the OMORIG agent. OMORIG looks up the object id in the Global Database and then routes this IOCTL to OMORI agent where found object lives. When IOCTL completed, an IOCTL reply is sent again to the OMORIG, which forwards this reply to originator of the IOCTL request. As seen from the above description with increasing number of the IOCTL requests, OMORIG agent becomes a bottleneck.

In one embodiment, to eliminate unnecessary traffic, an OMORI cache is designed. By this design OMORI maintains cache table of the objects IDs. When IOCTL is to be forwarded OMORI agent checks cache table. If object ID not found then IOCTL is forwarded to the OMORIG as in original scheme. When IOCTL reply is received object ID is inserted in the cache table. If object ID found the IOCTL is forwarded directly to OMORI, identified by the address space id saved in the object ID. Cache table is invalidated periodically.

In one embodiment, OMORI cache table is designed to use a *closed hashing* algorithm (also known as *open addressing*). In a closed hashing system, if collision occurs, alternate cells are tried until the empty cell is found. In one embodiment, closed hashing with linear probing is used. In one such embodiment, limited search is added such that, in case of collision only a limited number of cells will be tried. If empty cell is not found, then a new entry will replace the collided one.

In one embodiment, all elements in the OMORIG as well as in the OMORI database are managed using double linked circular list. As the number of elements in the list increases rises, however, the problem of search latency becomes an issue. In one embodiment, therefore, lists (which supposedly have large number of elements) are modified to the hash table. Open hashing is used for this purpose. Open hashing is to keep a list of all elements that hash to the same value.

One embodiment of a Control Blade Redundancy algorithm will be discussed. As noted above, in one embodiment, system 10 is designed to provide Fault Tolerance. In one such embodiment, each Control Blade runs management modules such as Command Line Interface (CLI) and Simple Network Management Protocol (SNMP), which allows configuration of system 10. Each of these modules retrieves data from the OM Global Database that resides on the Control Blade. Global database is constructed from the distributed OM Local Databases, which are stored on every processing node in the system.

In one embodiment, each switch 12 has at least two Control Blades. In the event of Control Blade failure, system management and configuration are done using the backup Control Blade. Thus NOS 20 provides a Control Blade Redundancy (CBR) service. This document discusses the protocol used to provide a synchronized backup of the OM Global Database as part of the CBR service.

In the following description,

Master – Processing Engine 0 (PE 0) on the Control Blade (CB) which is being use to manage and configure the system and participates in the message passing communication.

Slave – Processing Engine 1-3 (PE 1-3) on the CB and PE0-4 on Access or Processor Blades which participates in the message passing communication.

Standby **-** Processing Engine 0 (PE 0) on the Control Blade (CB) which can be used to manage and configure the system in the case of Master failure and participates in the message passing communication.

Peer – any Processing Engine which participates in the message passing communication

As noted above, NOS 20 consists of the several layers. Figure 24 shows only layers, which are relevant to the Database Redundancy problem that is discussed in this document.

Control Ring Driver 23 – notifies upper layer on the following events:

- blade up: new blade inserted in the system and became operational
- blade down: blade removed from the system and became non-operational;
- master blade up: new CB inserted and Control Ring decided that this is a Master
- standby blade up: new CB inserted and Control Ring decided that this is a Standby
- slave blade up: new blade inserted and this is not CB.

Distributed Messaging Layer (DML) 22 is message passing model to provide inter connectivity between processing nodes and channel management. DML 22 provides a reliable group communication based on message passing infrastructure by implementing:

- reliable sequenced layer
- supports of channels that provide independent communication universes
- Group operation on the channel like send, recv barrier synchronization and broadcast.
- Dynamic group membership that reflects dynamic state of the system.

Object Manager (OM) 24 is a module, which manages VPN, VR, objects and object groups in the system. Provides an IOCTL like mechanism for reliable fault tolerant messaging between objects that is typically used for management function. This uses the DML channel "DML_CHAN_WORLD". This mechanism was described above.

CB Channel 25 is a DML channel whose members are the dynamic set of Control Blades present and intercommunicating in the system.

Control Blade Redundancy (CBR) 26 is a module, which provides redundant Global Database on the Standby blades; CBR 26 is a DML application that receives notification from DML on all UP/DOWN events.

In one embodiment, Control Blade redundancy (CBR) 26 is designed to create and maintain replicas of the Master Control Blade management information on the Standby Control Blades and to reuse that information in the case of failure of the current Master and election of a new Master. Control Ring Driver 23 normally elects the new Master. If the Control Ring detection mechanism fails, however, a software-based leader election protocol implemented by DML 22 will elect the new Master. This redundancy is illustrated in Figure 25.

An important part of the management information is the OM Global Database. A key issue of the CBR is the consistency of OM Global Database. The OM Global Database is synchronized in two ways: bulk updates and flash updates. Bulk updates are used in CBR 26 on dynamic events like peer up/down. Flash updates are used to propagate individual change in the database (like a VR being created or deleted).

There are four Data Types which CBR protocol supports: Virtual Private Network (VPN), Virtual Router (VR), GROUP (an internal representation of the VR; set of all objects belonging to VR), and Object ID (OID).

CBR protocol provides sequential messaging per Data Type. If Standby receives update message with sequence number, which is not equal to the one expected then Standby sends message about it to Master and Master restarts update from the beginning. Note that DML provides a sequenced reliable transport and this should not happen normally. It could happen if the underlying SRTP Point-to-Point link resets as a result of timeout.

As a DML application CBR 26 is notified of events happening in the system. The events indicated are *peer up, peer down, master up, master down.*

On peer down event CBR does not need to take any action, OM on every Control Blade will update its database.

On master up/master down event CBR also does not need to take any action, because master up event always comes with peer up/peer down event where all the actions were taken.

On peer up event Master will dump its own Global Database to the all Standby Nodes. The dump algorithm is described in the Transition Diagram.

Figure 26 shows Master's State Transition Diagram. Master maintains state of each node participating in the CBR protocol. Master itself does not transition from the READY State. When peer up event occurs for a standby a new CBR node is added to the list and state is initialized to READY. In Figure 26, DT denotes Data Types (described above). ALL_MARKED is a bitmap that is used to keep track of the MARK replies for the specific Data Type. When all replies arrived bitmap is equal to bitmask, which means that all Data Types were marked.

ALL_FINISHED is a bitmap that is used to keep track of the FINISH replies for the specific Data Type. When all replies arrived bitmap is equal to bitmask, which means that all Data Types were finished

DUMP Master State Transition Table is given on the Table 4.

STATE	EVENT	ACTION
READY	PEER UP	Send MARK request for each Data Type Transit to START DUMP
READY	RECV DUMP request from Standby (invalid message sequence number)	Send MARK request for each Data Type, clear ALL_MARKED Transit to START DUMPs
vSTART DUMP	RECV MARK reply for one of the Data Types &&! ALL Data Types MARKED	Modify ALL_MARKED to include replied peer
START DUMP	RECV MARK reply for one of the Data Types && ALL Data Types MARKED	Transit to DUMP_IN_PROGRESS State. For each Data Type send

		DUMP DATA; For each Data Type send FINISH DATA; Transit to FINISH_DUMP;
START DUMP	RECV DUMP request from Standby (invalid message sequence number)	Send MARK request for each Data Type, clear ALL_MARKED
DUMP_IN_ PROGRESS	RECV DUMP request from Standby (invalid message sequence number)	Send MARK request for each Data Type Transit to START DUMP
FINISH DUMP	RECV FINISH reply for one of the Data Types && !ALL Data Types FINISHED	Modify ALL_FINISHED to include replied peer
FINISH DUMP	RECV FINISH reply for one of the Data Types && ALL Data Types FINISHED	Transit to READY
FINISH DUMP	RECV DUMP request from Standby (invalid message sequence number)	Send MARK request for each Data Type Transit to START DUMP

Table 4. DUMP Master State Transition Table.

Standby Node maintains the state transitions shown on Figure 27 and in Table 5 only for itself. When CBR 26 is initialized state is READY

STATE	EVENT	ACTION
READY	RECV MARK request for one of the Data Types	Send MARK reply for this Data Type Transits to START DUMP
READY	RECV invalid message sequence number for flash updates	Send DUMP request to Master
READY	RECV ADD request for one of the Data Types and invalid message sequence number	Send DUMP request to Master
READY	RECV ADD request for one of the Data Types	Add data to the Database
READY	RECV DELETE request for one of the Data Types and invalid message sequence number	Send DUMP request to Master
READY	RECV DELETE request for one of the Data Types	Delete data from the Database

START DUMP	RECV MARK request for one of the Data Types && ALL Data Types MARKED	Transit to DUMP_IN_PROGRESS State.
		For each Data Type send DUMP DATA;
		For each Data Type send FINISH DATA;
		Transit to FINISH_DUMP;

START DUMP	RECV invalid message sequence number	Send DUMP request to Master Transit to READY
DUMP_IN_ PROGRESS	RECV DUMP request for one of the Data Types	if this is a new item then add to Database, otherwise modify existing one.
DUMP_IN_PROG RESS	RECV invalid message sequence number	Send DUMP request to Master Transit to READY
FINISH DUMP	RECV FINISH request for one of the Data Types && !ALL Data Types FINISHED	Modify ALL_FINISHED to include replied peer Send FINISH reply for this Data Type
FINISH DUMP	RECV FINISH request for one of the Data Types && ALL Data Types FINISHED	Send FINISH reply for this Data Type Transit to READY
FINISH DUMP	RECV invalid message sequence number	Send DUMP request to Master Transit to READY

Table 5. Dump Standby State Transition Table

Master sends MARK request for each data type to this peer and transits to the START_DUMP state. When Standby receives mark request for one of the data types it transits to START_DUMP state, marks all existing elements of specified type and sends reply back to the Master. In its turn master delays start of dump until it receives MARK replies for all the Data Types. When all the replies are received Master transits to DUMP_IN_PROGRESS state and dumps all elements of its Database to the Standby peer. Standby receives DUMP message and updates its data in the Database and unmarks updated element. When DUMP is done Mater sends to Standby FINISH message and transits to the FINISH_DUMP state. After receiving FINISH message Standby transits to the FINISH_DUMP state, deletes all the elements in the Database, which are left marked and sends FINISH reply to the Master. Standby stays in this state until finish procedure done for all Data Types and then goes into READY STATE. Master remains in the FINISH state until FINISH replies are received for all Data Types. If Standby receives message with invalid sequence number it sends DUMP_REQUEST to the master and transits to READY state from the state where Standby was when message arrived. Upon receiving DUMP_REQUEST Master transits to START_DUMP state.

OMORIG on Master blade calls CBR 26 to update all known Standby Peers for every change in the Global Database. There are two types of changes: ADD and DELETE. When Standby receives ADD update it looks up in its replicated database for a requested data type by the specified ID. If the specified data item is found then it is modified with received information. If search fails then new data item is created and added to the database. On the DELETE request Standby finds requested data type and removes it without removing semantic dependencies. The OM on Master observes all semantic dependencies when it calls CBR to update a particular Data Type.

Standby Peer maintains simple FSM for flash updates as shown on Figure 27.

Flash Updates as well as Bulk updates are sequential and loss of a message causes restart of the Dump procedure.

A representative CBR API and associated data structures is shown in Appendix B.

In one embodiment, to be absolutely sure that Standby OM Global Database is a mirror from the Master OM Global Database, periodic updates are used. Standby can run periodic update infrequently. Periodic update is based on the consistency rules checks. If one of the consistencies rules fails, then Standby requests Bulk update from the Master.

Consistency rules for OM Global Database are:

- For every VPN there is a unique ID in the system.
- For every VR there is a unique combination of the VPN ID and VR ID.
- Every VR has a unique ID in the scope of VPN.
- For every group there is a VR to which this group belongs.

- For every object there is a group to which this object belongs.
- Every object has a unique ID in the system.
- Value of counter "Number of VRs in the VPN descriptor "is equal to total number of VRs per VPN.
- Value of the counter "Number of objects in the group descriptor" is equal to the total number of objects in the group.
- Total number of objects across all groups is equal to the total number of objects across address spaces, and it is equal to total number of objects in the system.
- Number of objects in the sorted global list is equal to number of objects in the global hash, and it is equal to total number of objects in the system.
- For every object class in the system there is a corresponding entry in the class table.

In one embodiment, the service provider's security staff consults with the customer in order to understand the corporate network infrastructure and to develop appropriate security policies (Note: this is a similar process to the CPE model). Once this has been accomplished, the NOC security staff remotely accesses the IP Service Processing Switch (using the Service Management System) at the regional POP serving the enterprise customer, and the firewall service is provisioned and configured remotely.

Conclusion

System 10 as described above enables the service provider to leverage the enterprise's existing services infrastructure (leased lines and Frame Relay PVCs) to deliver new, value-added services without the requirement of a truck roll. All firewall and VPN functionality resides on the IP Service Processing Switch at the POP, thus freeing the service provider from onsite systems integration and configuration and effectively hiding the technology from the enterprise customer. Firewall inspection and access control functions, as well as VPN tunneling and encryption, take place at the IP Service Processing Switch and across the WAN, while the enterprise's secure leased line or Frame Relay PVC access link remains in place. The customer interface is between its router and the IP Service Processing Switch (acting as an access router), just as it was prior to the rollout of the managed firewall service. Additionally, the customer has visibility into and control over its segment of the network via the CNM that typically resides at the headquarters site.

The network-based firewall model also enables service providers to quickly and cost-effectively roll out managed firewall solutions at all enterprise customer sites. As a result, secure Internet access can be provided to every site, eliminating the performance and complexity issues associated with backhauling Internet traffic across the WAN to and from a centralized secure access point. As the IP Service Delivery Platform is designed to enable value-added public network services, it is a carrier-grade system that is more robust and higher-capacity than traditional access routers, and an order of magnitude more scaleable and manageable than CPE-based systems. The platform's Service Management System enables managed firewall services, as well as a host of other managed network services, to be provisioned, configured, and managed with point-and-click simplicity, minimizing the need for expensive, highly skilled security professionals and significantly cutting service rollout lead-times. The Service Management System is capable of supporting a fleet of IP Service Processing Switches and tens of thousands of enterprise networks, and interfaces to the platform at the POP from the NOC via IP address. Support for incremental additional platforms and customers is added via modular software add-ons. Services can be provisioned via the SMS system's simple point and click menus, as well as requested directly by the customer via the CNM system. Deployment of a robust IP Service Delivery Platform in the carrier network enables service providers to rapidly turn-up high value, managed network-based services at a fraction of the capital and operational costs of CPE-based solutions. This enables service providers to gain a least-cost service delivery and support structure. Additionally, it enables them to gain higher margins and more market share than competitors utilizing traditional service delivery mechanisms — even while offering managed firewall services at a lower customer price point.

As enterprise customers gain confidence in the WAN providers' ability to deliver managed firewall services, a more scaleable and cost-effective service delivery model must be employed. Moving the intelligence of the service off of the customer premises and into the WAN is an effective strategy to accomplish this. Managed, network-based firewall services provide the same feature/functionality of a CPE-based service while greatly reducing capital and operational costs, as well as complexity.

The managed, network-based firewall service model enables WAN service providers to minimize service creation and delivery costs. This model virtually eliminates the need for onsite installation, configuration, and troubleshooting truck rolls, drastically reducing operational costs. This lower cost structure creates opportunities to increase revenues and/or gain market share by value-pricing the service. Services can be rapidly provisioned via a centralized services management system, shortening delivery cycles and enabling service providers to begin billing immediately. Additional services can be rapidly crafted and deployed via the same efficient delivery mechanism.

The network-based service model is a rapid and cost-effective way for service providers to deploy high-value managed firewall solutions. This model requires a comprehensive service delivery platform consisting of robust network hardware, an intelligent and scaleable services management system, and a feature-rich Customer Network Management (CNM) tool to mitigate customers' fears of losing control of network security.

In the above discussion and in the attached appendices, the term "computer" is defined to include any digital or analog data processing unit. Examples include any personal computer, workstation, set top box, mainframe, server, supercomputer, laptop or personal digital assistant capable of embodying the inventions described herein.

Examples of articles comprising computer readable media are floppy disks, hard drives, CD-ROM or DVD media or any other read-write or read-only memory device.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiment shown. This application is intended to cover any adaptations or variations of the present invention. Therefore, it is intended that this invention be limited only by the claims and the equivalents thereof.

What is claimed is:

1. A method of managing a switch, comprising:

installing the switch having a plurality of processor elements;

installing an operating system on each processor element;

creating a system virtual router;

configuring the processor elements from the system virtual router.

2. An article comprising a computer readable medium having instructions thereon, wherein the instructions, when executed in a computer, create a system for executing the method of claim 1.

3. A switch management system, comprising:

an object manager; and

a distributed management layer, wherein the object manager communicates with objects through the distributed management layer.

SWITCH MANAGEMENT SYSTEM AND METHOD

Abstract

A system and method of managing a switch includes installing a switch having a plurality of processor elements, installing an operating system on each processor element, creating a system virtual router and configuring the processor elements from the system virtual router.

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 C.F.R. 1.10 on the date indicated below and is addressed to the Assistant Commissioner for Patents, Attn: Box Patent Application, Washington, D. C. 20231.
"Express Mail" mailing label number EL618477208US
Date of Deposit September 13, 2000
Printed name Charles A. Lemaire
Signature [illegible] 13 Sept 2000

CoSine IP Service Delivery Platform Application Architecture

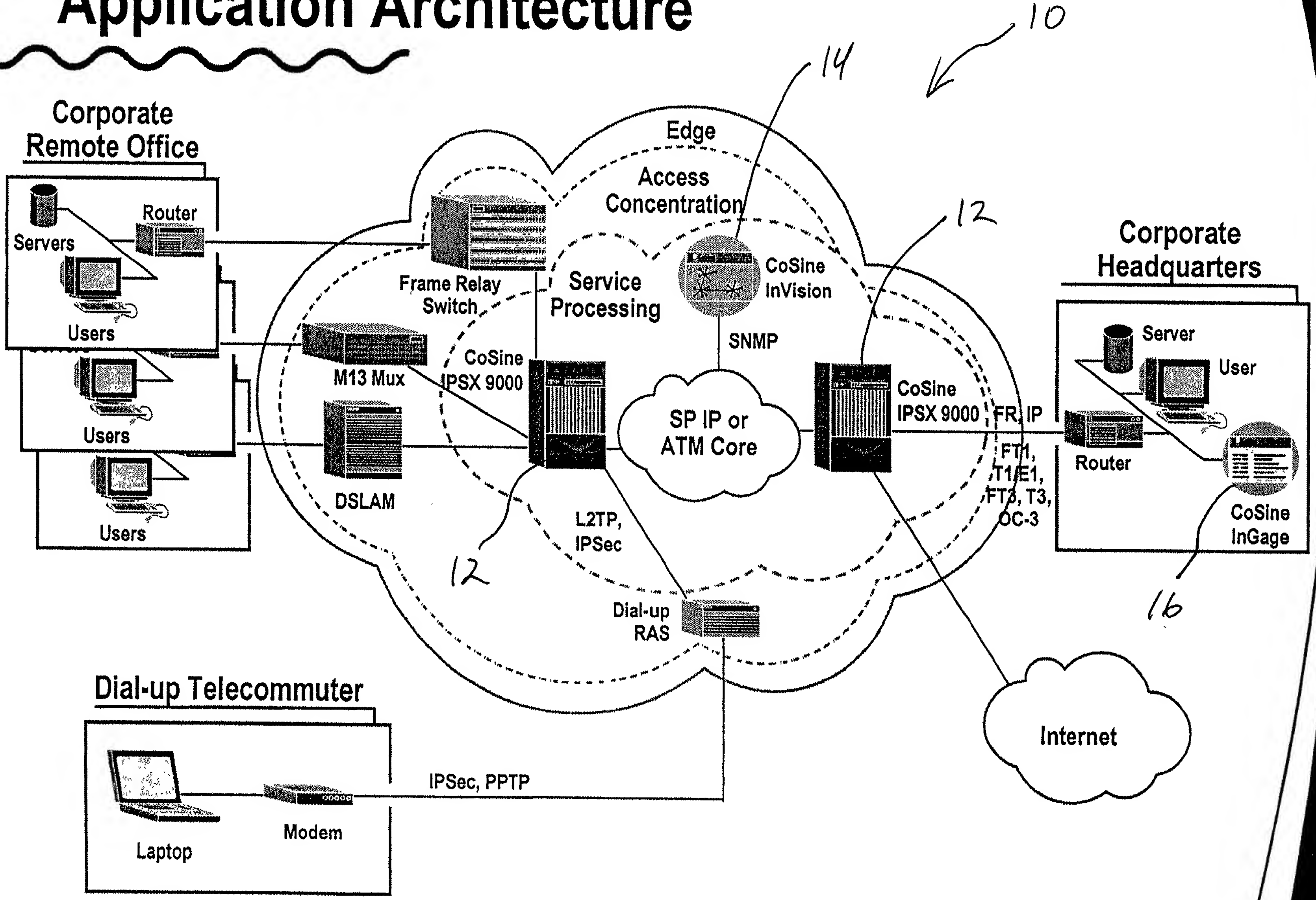
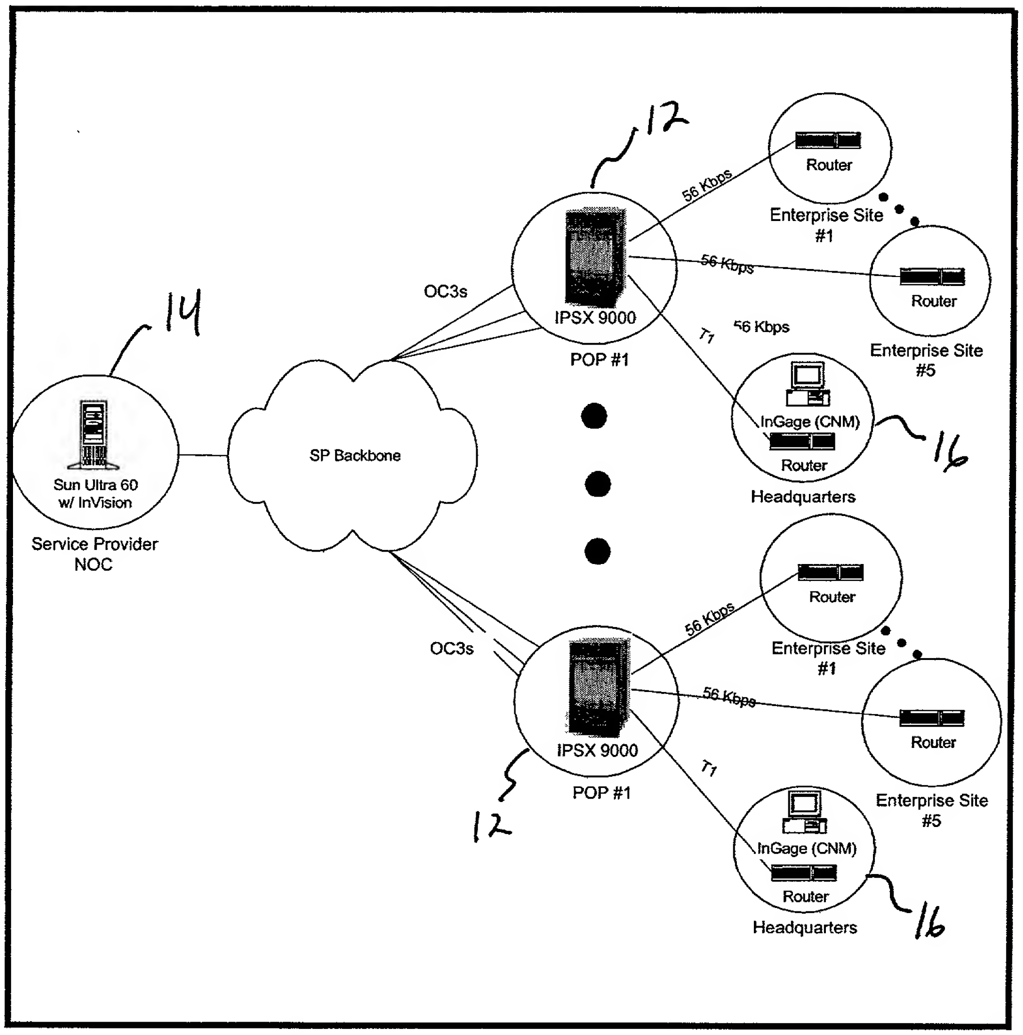


Figure 1

Diagram 5 — Managed Firewall Service with CoSine's Network-based Solution



POP Infrastructure

The POP access infrastructure in the network-based managed firewall service model is based on the CoSine Communications IPSX 9000 Service Processing Switch. The base configuration for the switch includes:

- 26-slot chassis
- Redundant power supply
- IPNOS Base Software
- Ring Bridge & Ring Bridge Pass-Thru (to complete midplane)
- Control Blade (for communications with InVision Services Management System)
- Dual-port Channelized DS3 Access Blade
- Dual-port Unchannelized DS3 Access Blades
- Processor Blade
- OC-3c POS Trunk Blade

The following tables analyze the cost structure of all of the above models and projects these costs out over 5 years:

Figure 2

IPNOS Overview

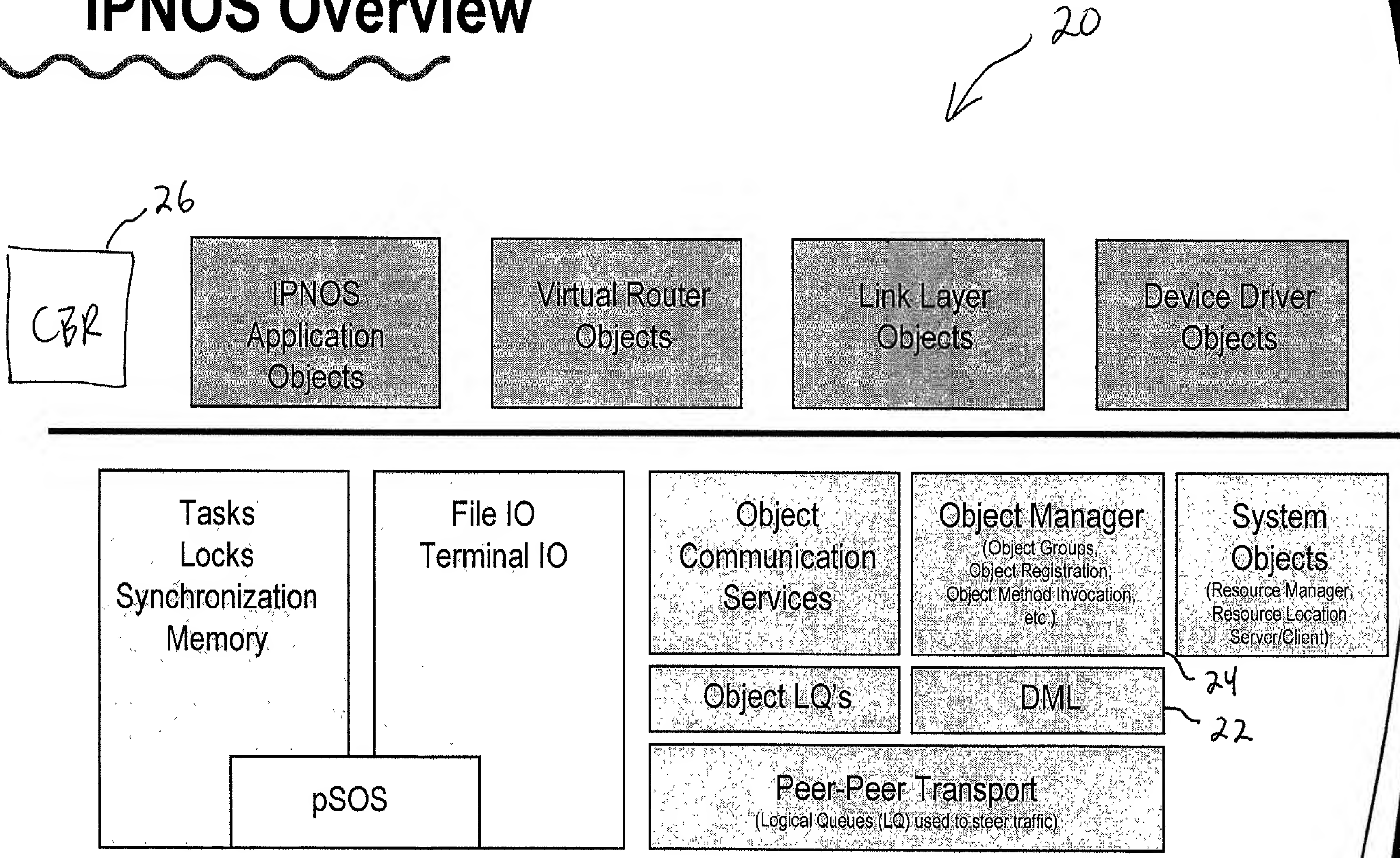

20
26
CBR
IPNOS Application Objects
Virtual Router Objects
Link Layer Objects
Device Driver Objects
Tasks
Locks
Synchronization
Memory
File IO
Terminal IO
pSOS
Object Communication Services
Object Manager
(Object Groups, Object Registration, Object Method Invocation, etc.)
System Objects
(Resource Manager, Resource Location Server/Client)
24
Object LQ's
DML
22
Peer-Peer Transport
(Logical Queues (LQ) used to steer traffic)

Fig. 3

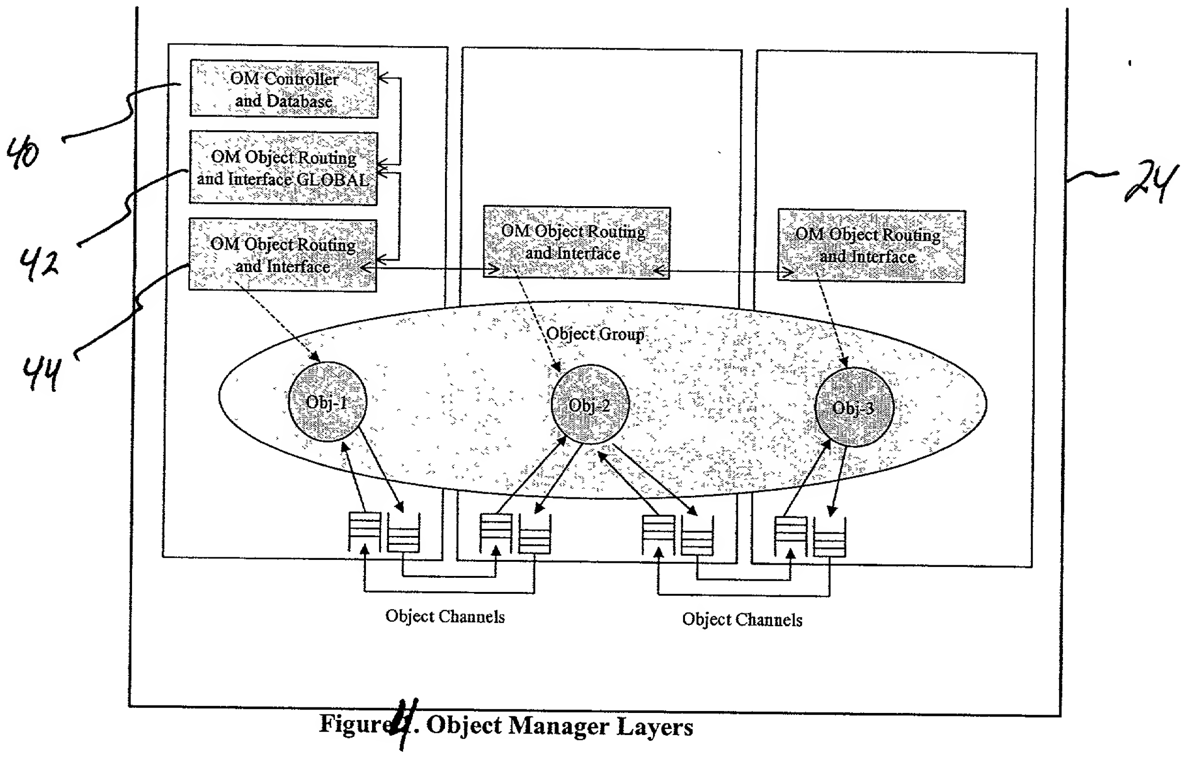

OM Controller and Database
OM Object Routing and Interface GLOBAL
OM Object Routing and Interface
OM Object Routing and Interface
OM Object Routing and Interface
Object Group
Obj-1
Obj-2
Obj-3
Object Channels
Object Channels
40
42
44
24

Figure 4. Object Manager Layers

IPSX object database consists of two types of database: Global, managed on Master Control Blade by OMORIG and distributed local databases, managed by OMORI agents on every PE present in the system. Global database is a superset of the extracts from local databases.

2.2. Object

Objects represent a basic unit of management for purposes of fault tolerance, computational load balancing etc. One or more adjacent protocol modules can be placed into a single object. It is also possible that a module is split across two objects.

2.3. Group

Group is an aggregation point for all objects that comprises the VR. Group and VR have one-to-one mapping. A Group encompasses objects, which are located in different address spaces. Group Id, which identifies a group, is unique in the scope of a single IPSX system.

2.4. Object Class

Figure 5 shows the distinction between an Object Class and an Object Group. Both are collections of objects. As shown in Figure 2, an object class is a set of objects that have the same type signature and behavior (e.g. Applications Class, TCP/IP Class and Interfaces Class). In contrast, for an object group, the constituent objects do not necessarily have the same type signature and behavior (e.g. Object Groups 1 to 3). There can be multiple objects of the same class in an object group (e.g. Object Group 2 has two objects of Interface Class). On the other hand, an object group need not have an object of each class (e.g. Object Group 3 does not have an object of Interface Class).

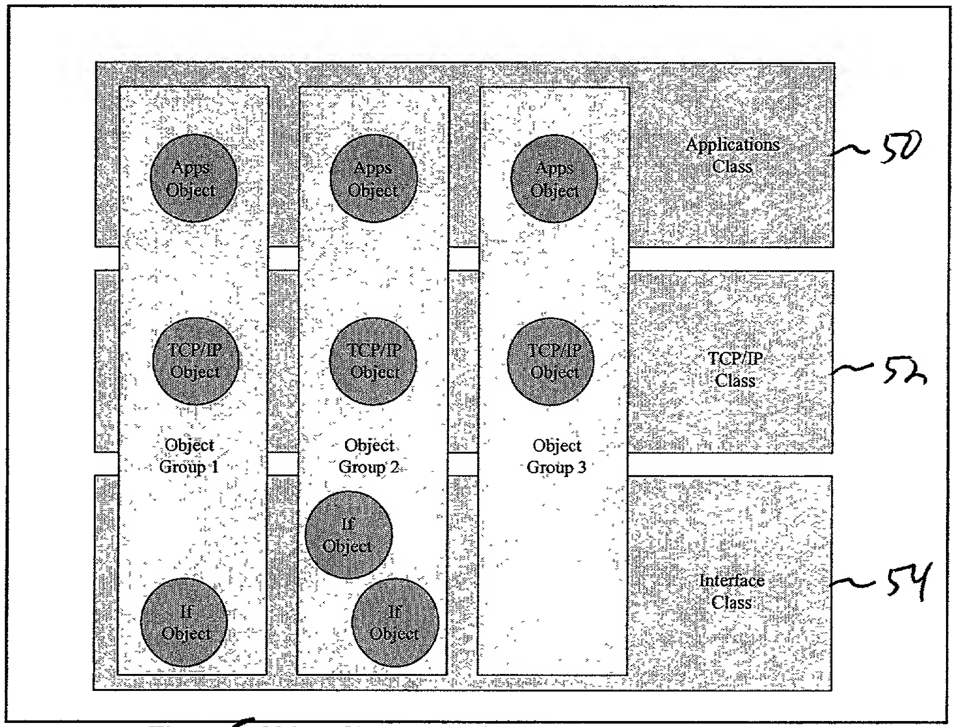


Figure 5. Object Classes and Groups.

2.5. OMCD and OMORIG

OMCD is the agent, which interfaces to the Configuration Manager. As shown on Figure 3 OMCD manages

- Global list of VPN in the IPSX system
- List of VRs per VPN

The caveats for VPN and VRs are:

- VPN ID equal to 0 is illegal;
- Global uniqueness of VPN ID across IPSX systems is the responsibility of the Service Management System (SMS).

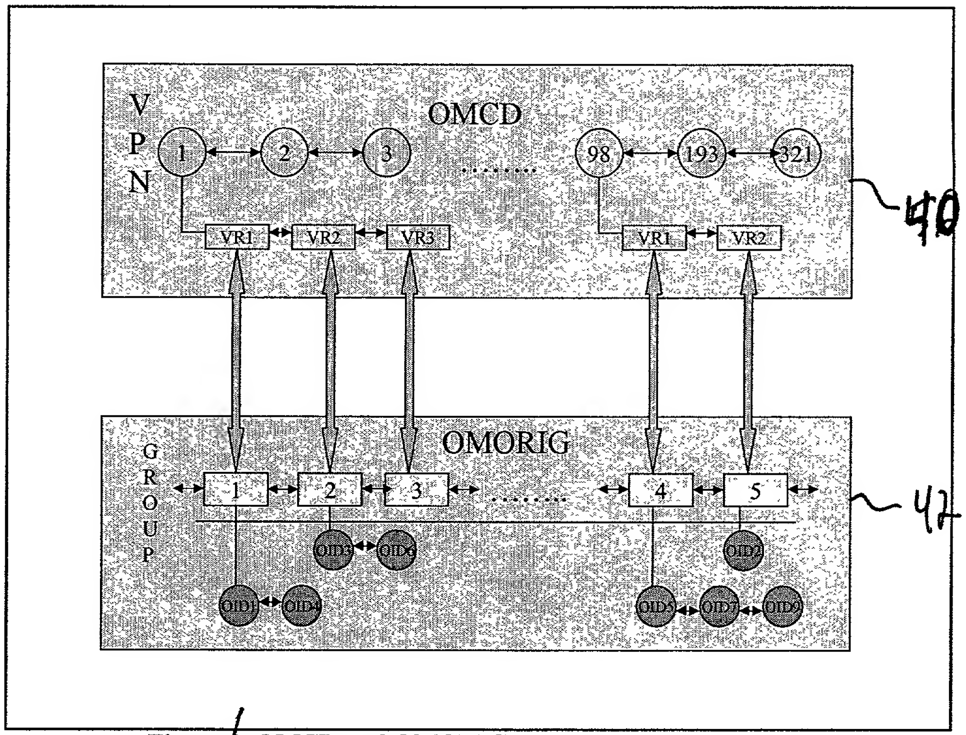


Figure 6 OMCD and OMORIG Database maps.

OMCD creates vpn descriptor every time when Configuration managers request VPN creation. Every VPN is identified by unique VPN ID.

Virtual Router (VR) is identified by VR ID, which is the IP Address of the VR. VR ID is unique in the VPN context. When Configuration Manager requests creation of an existing VR in the VPN, VR creation request is rejected. Otherwise vr descriptor will be created.

There are several types of the VR:

- ISP (Internet Service Provider) VR: Typically there is 1 such VR for a single IPSX.
- Control VR: There can be only one Control VR for a single IPSX. This VR is used to host the management applications such as SNMP, Telnet etc.
- Customer VR: There are several Customer VRs in a single IPSX. Typically, there is 1 Customer VR per customer service point.

Detailed VR creation process is described below

OMORIG agent runs on every Control Blade, whether it is Master or Standby Blade. OMORI local sends the change only to Master. Control Blade Redundancy feature, described in [4] takes care of replicating and synchronizing OMORIG database from Master to Standby.

2.6. OMORIG Object ID Links

OMORIG provides several mappings of Object Ids. It manages list of object Ids, which

- Are located on the same address space.
- Belong to the same group
- Sorted Global object ID list
- Unsorted Global object ID list

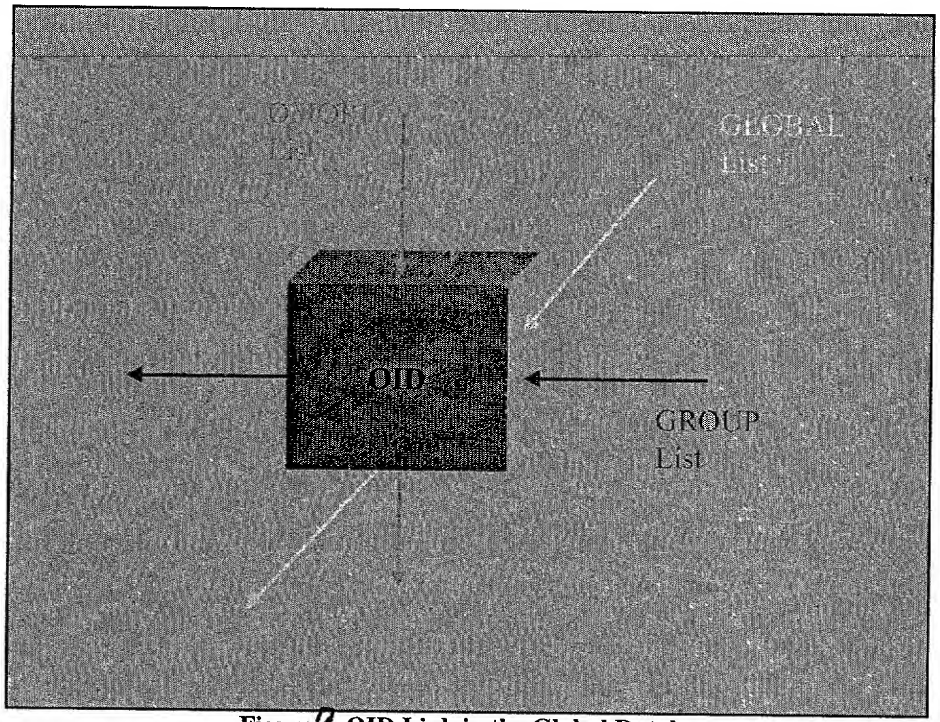


Figure ~~4~~7. OID Link in the Global Database

OID link data structure and QMORIG API are published in page 40.

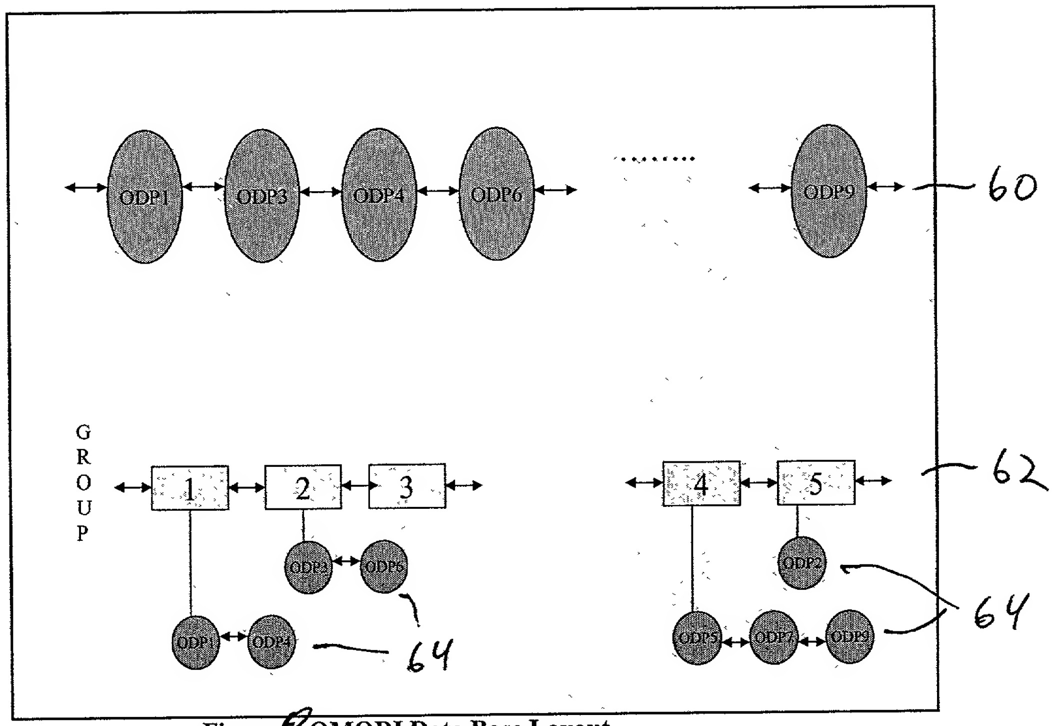


Figure 8 OMORI Data Base Layout

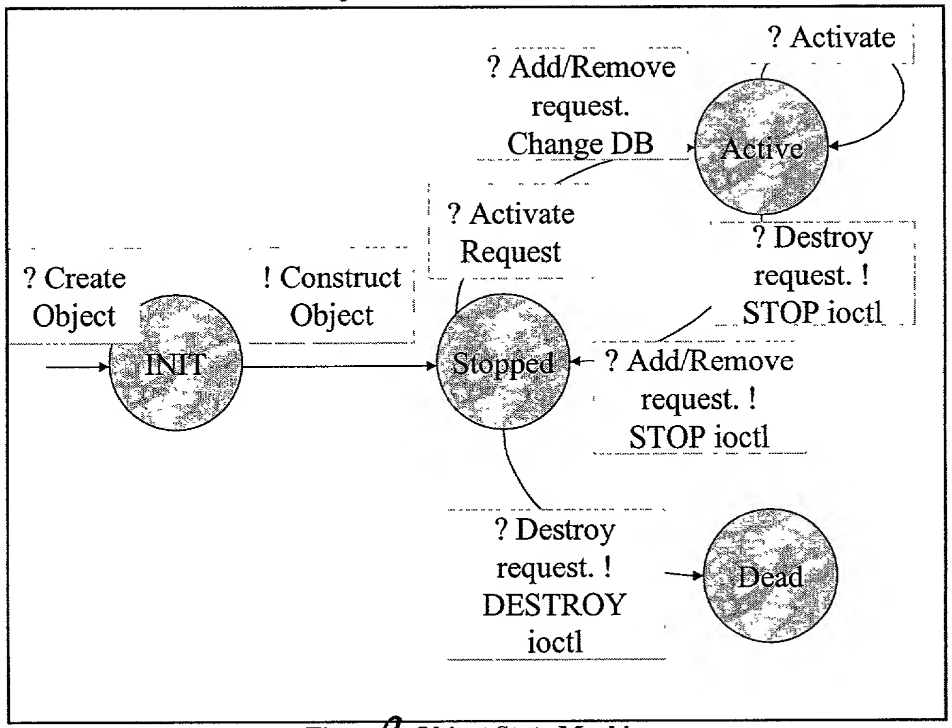

? Activate
? Add/Remove request. Change DB
Active
? Activate Request
? Create Object
! Construct Object
? Destroy request. ! STOP ioctl
INIT
Stopped
? Add/Remove request. ! STOP ioctl
? Destroy request. ! DESTROY ioctl
Dead

Figure 9. Object State Machine

3. OM as a DML Application

Distributed Messaging Layer (DML) is used to provide inter-processor communication and isolated channels for data and control messages. Detailed explanation on DML can be found in [3]. OMORIG and OMORI communicate via predefined DML channel DML_CHAN_DATA. All IPNOS nodes in the system are members of DML_CHAN_DATA. During initialization process OMORI register to DML receive function, which will be called every time a new packet arrives on DML_CHAN_DATA. OMORIG and OMORI are DML applications and therefore they are notified on every dynamic event in the system.

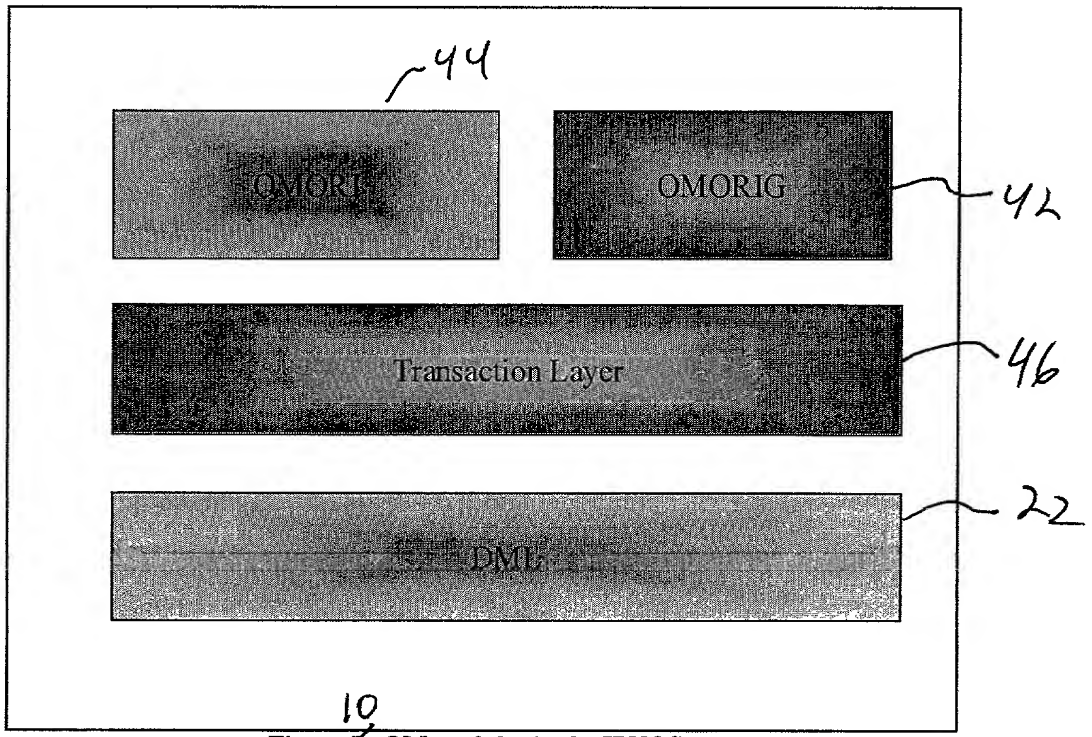


Figure ~~1~~. OM modules in the IPNOS.

3.1. Dynamic events

There are four types of dynamic events indicated by DML. These are:

- Peer Up – new IPNOS node detected and reachable.
- Peer Down – existing IPNOS node became unreachable
- Master Up – new Master elected in the system
- Master Down – existing Master became unreachable

On peer down event OMORI agent aborts all the pending transactions associated with the peer, which went down. J

3.1.2. OMORIG

On peer down event OMORIG destroys in its database all the objects, which are local to peer which went down. After that a scrub of the database is done. This includes destroying all groups, which do not have any objects in them and destroying VR associated with the group..

On peer up event and master up event OMORIG agent runs global database update protocol described in the section 3.1.3.

3.1.3. Update protocol

On peer up event OMORIG agent initiates a database update for local objects of the new peer. OMORIG maintains state machine per OMORI. Global Database Update State Transition Diagram is shown in Figure 8 and a detailed description of the transitions is in Table 2.

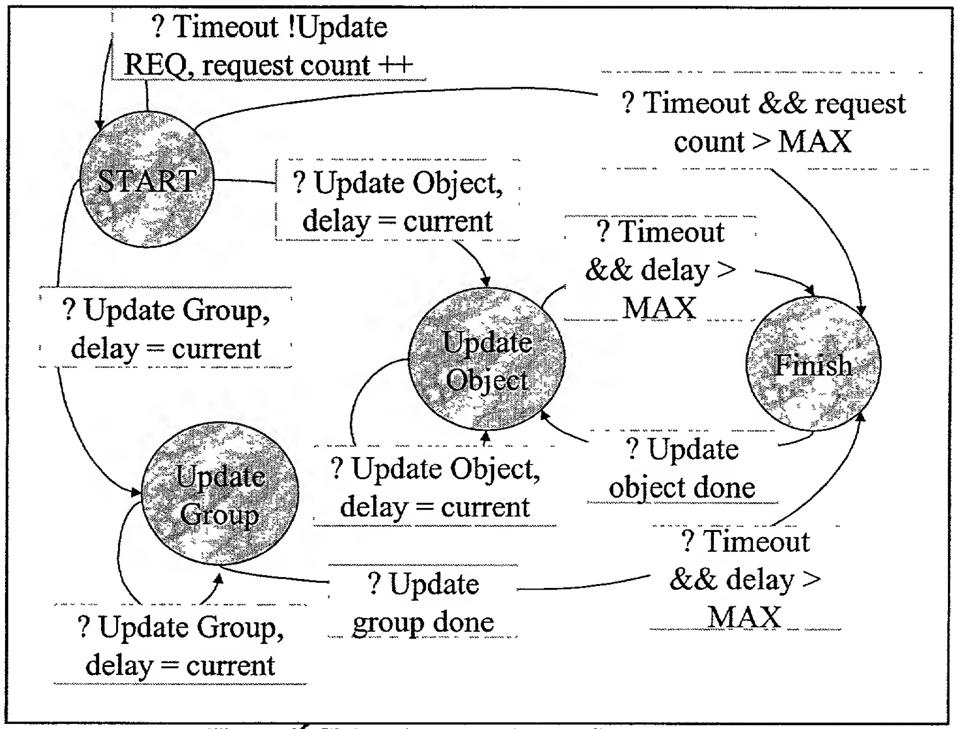

? Timeout !Update REQ, request count ++
? Timeout && request count > MAX
START
? Update Object, delay = current
? Timeout && delay > MAX
? Update Group, delay = current
Update Object
Finish
Update Group
? Update Object, delay = current
? Update object done
? Timeout && delay > MAX
? Update group done
? Update Group, delay = current

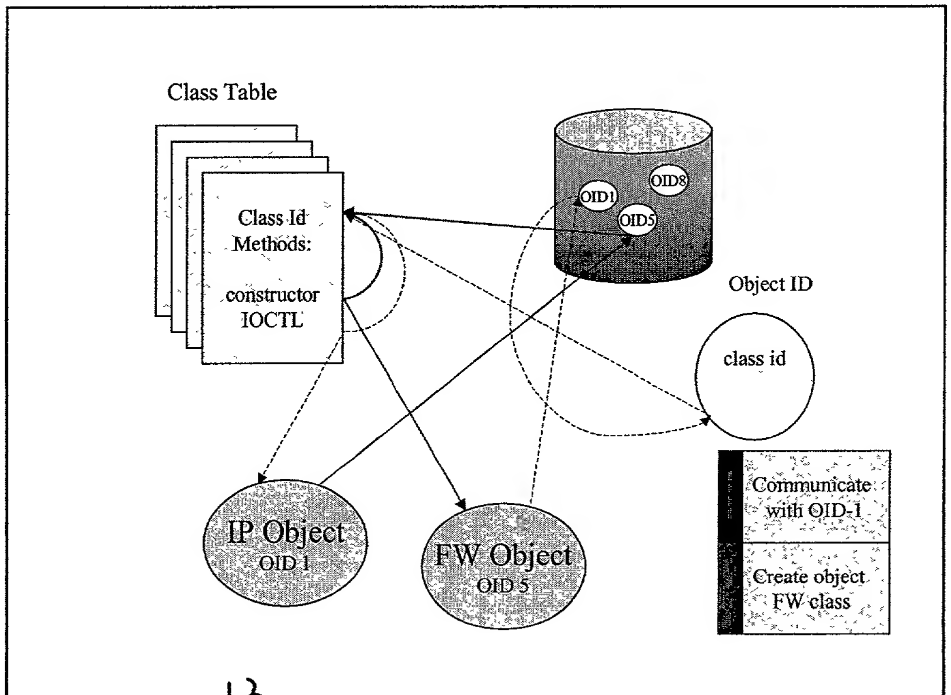
Figure ~~8~~. Global database Update State Machine
11

STATE	EVENT	ACTION
START	TIMEOUT && (request count < MAX)	Send update request
START	TIMEOUT && (request count > MAX)	Peer did not reply. Update FAILED Transit to FINISH state.
START	RECV UPDATE GROUP message	Transit to UPDATE GROUP state. Set last update equal to the current time.
START	RECV UPDATE OBJECT message	Transit to UPDATE OBJECT state. Set last update equal to the current time.

4. Objects: Creation and Inter Communication

4.1. Overview

In Figure 9 the communication for object creation is shown. IP object with OID 1 requests Firewall object to be created. OM creates object descriptor and based on the specified class of the object (e.g. Firewall), OM finds the appropriate *constructor* function in the object class table and constructs the new object. The same algorithm is used for management communications between objects (IOCTL). Based on the class id of the destination object appropriate *control* function from the object class table is called. It is the responsibility of the object implementers is to supply handlers for all supported IOCTLs.



12
Figure ~~9~~. Object Creation and Communication.

The IOCTL mechanism is described in section 4.3. Typically IOCTL between objects is used for Management Plane communications. For Data Plane communications between objects, object to object channels are used. Object channels are described in the section 4.5.

4.2. Creation

Objects can be created in three different ways:

- **REGISTERED**: Created as system comes up (e.g. drivers) and registered to the OMORI with object id, having physical location meaning.
- **CREATED BY OM**: Created by Object Manager. In this case OMORI creates a locally unique (in the scope of this address space) object ID that in conjunction with address space id gives unique object id inside of the system. To create an object, the constructor function based on the object class will be called.

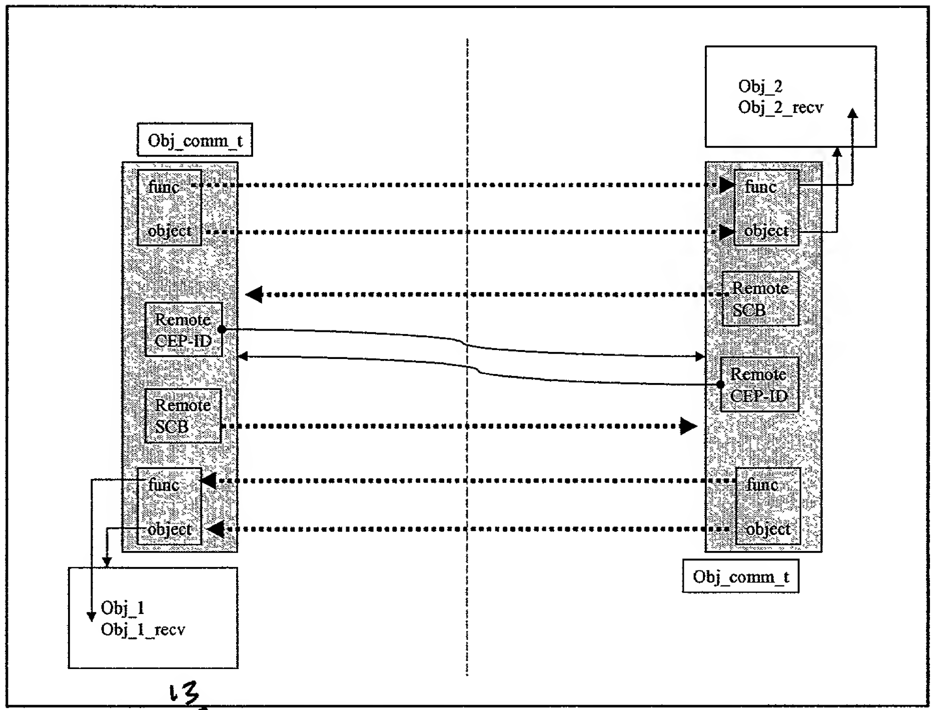


Figure ~~10~~ 13. Object Channels: Connecting CEPs in same address space

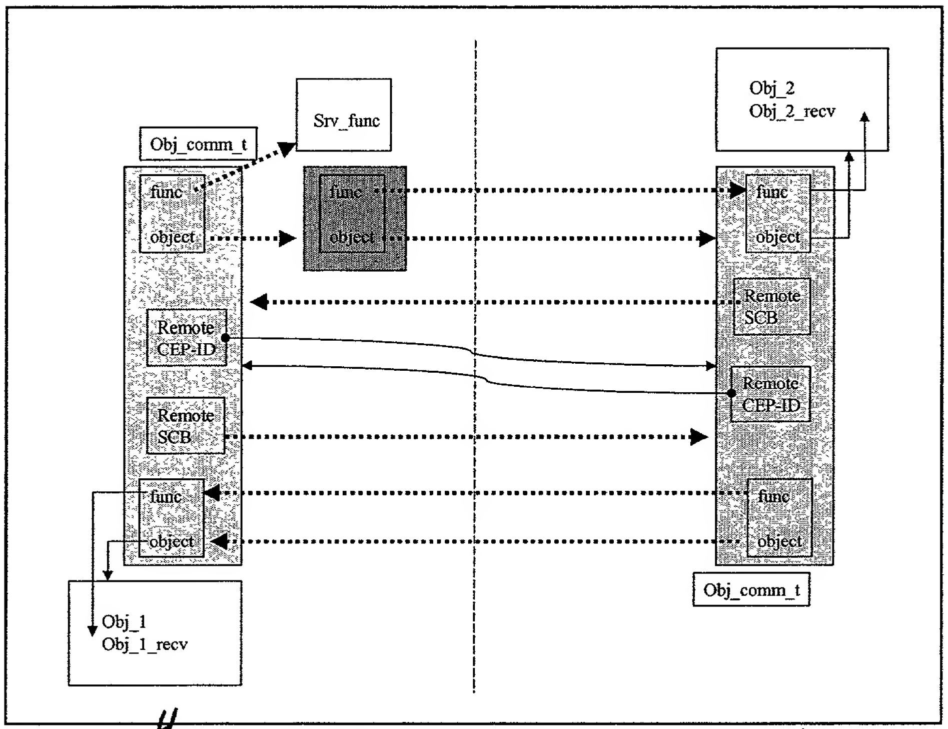


Figure 1~~1~~4: Object Channels: Pushing service on Transmit side of Object-1

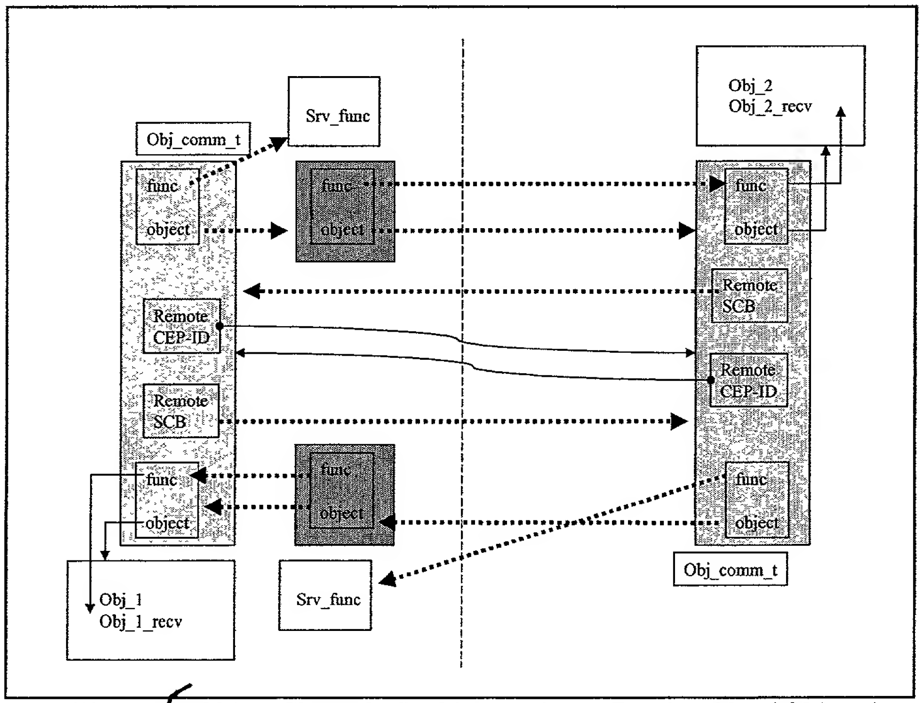


Figure 1~~2~~5: Object Channels: Pushing service on Receive side of Object-1

Step	Local CEP Object	Local IPNOS	Local RM/ LQ	Remote RM/ LQ	Remote IPNOS	Remote CEP Object
1	`obj_associate_channel(local_chan, local_cep_id, remote_cep_id)`					
2		***/* Allocate remote LQ */*** resmng_alloc_resource (RESOURCE_DATA_CONNECTION, 0, remote_cep_id -> object.address_space_id, &remote_lq)				
3				**Lookup resource tag and allocate from *remote* LQ**		
4		***/* Ask remote LQ to allocate local LQ */*** status = omori_obj_ioctl_by_id (&remote_lq, remote_lq.group, OBJ_CTL_CODE_ANY (LQUSER_BIND), &lq_bind, sizeof (lq_bind)); memcpy (&local_lq,				

		&lq_bind.lq_object.local, sizeof (object_id_t));				
5				**Use resmng_alloc_resource() to allocate *local* LQ**		
6			**Lookup resource tag and allocate from *local* LQ**			
7				**Return allocated *local* LQ**		
8		***/* Bind Local and Remote LQs */*** status = omori_obj_ioctl_by_id (&local_lq, local_lq.group, OBJ_CTL_CODE_ANY (LQUSER_BIND), &lq_bind, sizeof (lq_bind));				
9			**Setup LQ-API parameters to point to *remote* LQ**	**Setup LQ-API parameters to point to *local* LQ**		
10		***/* Push local LQ as a service onto local channel */*** status = omori_obj_ioctl_by_id				

Fig. 16b

		(&local_lq, local_lq.group , OBJ_CTL_CODE_ANY (LQUSER_BIND), &lq_bind, sizeof (lq_bind));				
11	**Lookup CEP address**					
12					***/* Push remote LQ as a service onto remote channel */*** status = omori_obj_ioctl_by_id (&remote_lq, remote_lq.group, OBJ_CTL_CODE_ANY (LQUSER_BIND), &lq_bind, sizeof (lq_bind));	
13						**Lookup CEP address**

Fig. 16c

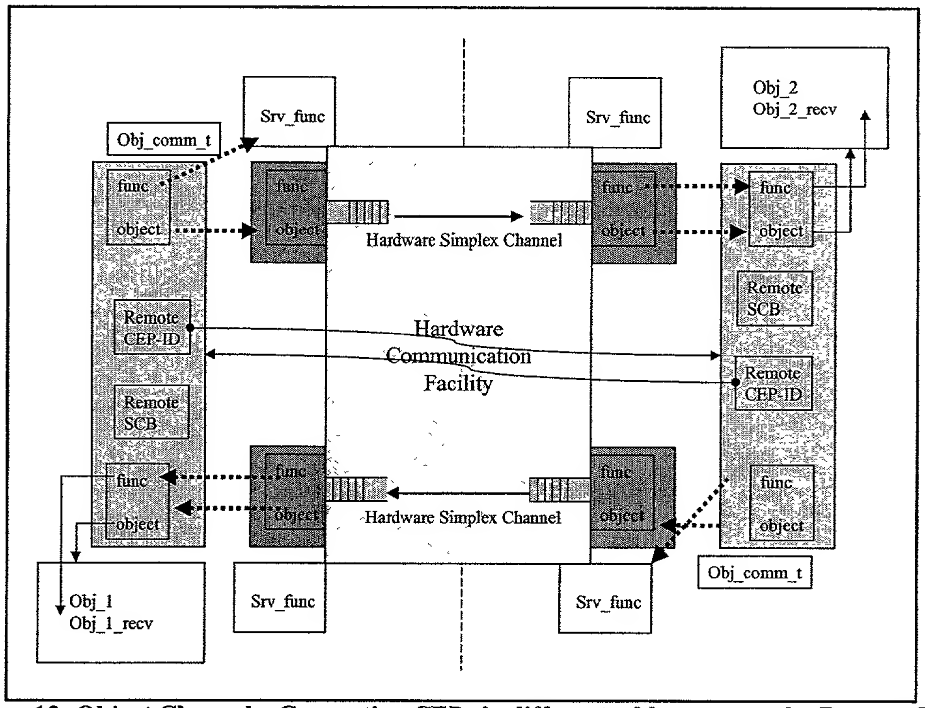


Figure ~~13~~ 17: Object Channels: Connecting CEPs in different address spaces by Remote Channel Service

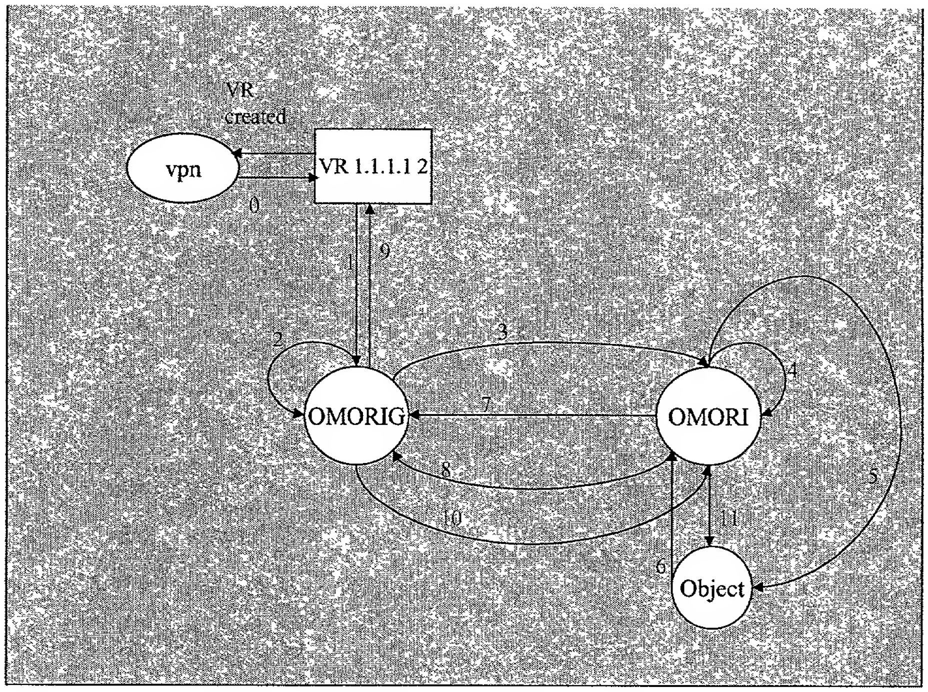


Figure 18 VR Creation with Single object

2		Create group 1; create object id link of selected class. Validate address space id on capability to service specified object class. Send request CREATE_OBJECT to capable OMORI (2). Wait for OMORI reply.		
3			Receive CREATE_OBJECT request for specified group. Lookup a group; not found. Create group 1; Create object descriptor of selected class. Call the class constructor.	
4		Receive MV_OBJ_TO_GROUP request; add object id to OMORIG Database	add object to the group, send MV_OBJ_TO_GROU P request to OMORIG	
5				Create and initialize an object; return SUCCESS or FAILURE.
6			If FAILURE remove object from the group, send MV_OBJ_TO_GROU P and RM_OBJ_FROM_GR OUP to OMORIG; ============== Else send reply for CREATE_OBJECT request to OMORIG	
7		Receive MV_OBJ_TO_GROUP request, move object to the group 0(OM_BASE_GROUP); Receive RM_OBJ_FROM_GROU P request; remove object id from OMORIG		

Fig. 19b

		Database ===============		
8		Receive Object CREATE reply. Signal to OMCD		
9	VR created, return status to user			
10		Send ACTIVATE object message to OMORI (2)		
11			Receive ACTIVATE object message. For all the objects of this group send generic IOCTL ACTIVATE_OBJECT	Activate object, Do object-specific action to make it active, operational

Fig. 19c

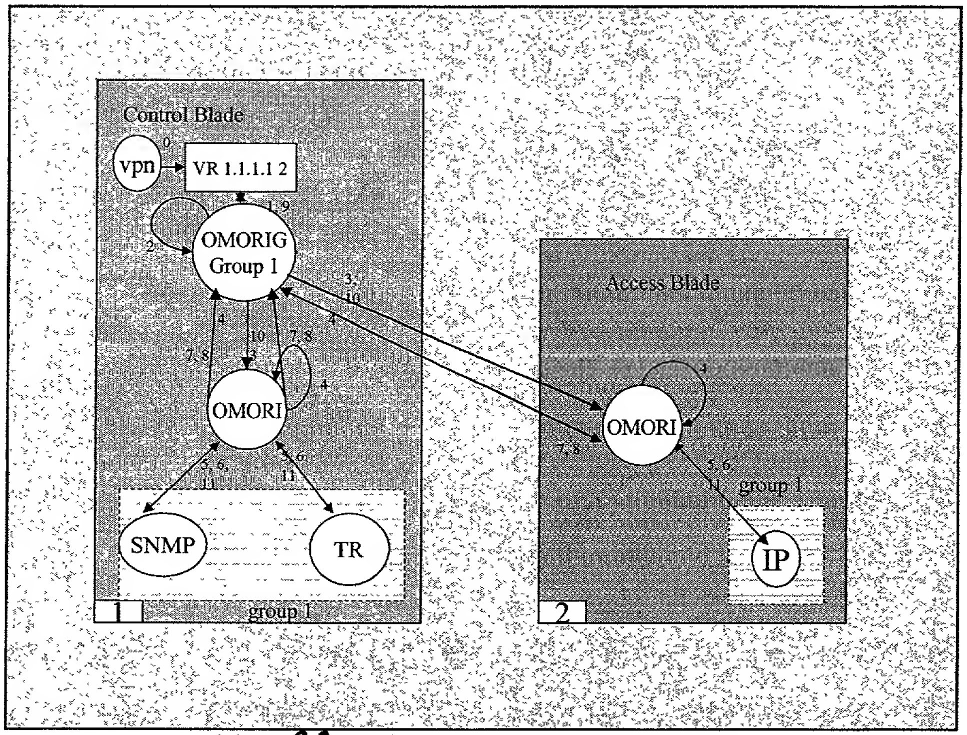


Figure 20 VR Creation with Multiple Objects.

Step	OMCD	OMORIG	OMORI	Object

0	Create unique vr_descriptor_t for specified VPN, fills with default value and adds VR to the list of VR per VPN.			
1	Requests group creation for specified VR with class_group_selector on specified address space.			
2		Create group 1; create object id link of selected class. Validate address space id on capability to service specified object class. Send request CREATE_OBJECT to capable OMORIs (1 and 2). Wait for reply from both OMORIs.		
3			Receive CREATE_OBJECT request for specified group. Lookup a group; not found. Create group 1; Create object descriptor of selected class. Call the class constructor.	
4		Receive MV_OBJ_TO_GROUP request; add object id to OMORIG Database	add object to the group, send MV_OBJ_TO_GROU P request to OMORIG	
5				Create and initialize an object; return SUCCESS or FAILURE.
6			If FAILURE remove object from the group, send MV_OBJ_TO_GROU P and RM_OBJ_FROM_GR OUP to OMORIG;	

Fig 21a

			================ Else send reply for CREATE_OBJECT request to OMORIG	
7		Receive MV_OBJ_TO_GROUP request, move object to the group 0(OM_BASE_GROUP); Receive RM_OBJ_FROM_GROUP request; remove object id from OMORIG Database ================		
8		Receive Object CREATE reply. If all the object replied then signal to OMCD, otherwise do nothing		
8	VR created, return status to user			
10		Send ACTIVATE object message to every OMORI (1,2) where objects were created		
11			Receive ACTIVATE object message. For all the objects of this group send generic IOCTL ACTIVATE_OBJECT	
12				Activate object, Do object-specific action to make it active, operational

Fig 21b.

5.2. VR Deletion

5.2.1. Multiple Objects

Compliment operation to create VR with multiple objects in the group is to destroy such a VR. Destroy VR operation is shown on the Figure 16. Sequence of steps, describing this is in the Table 7.

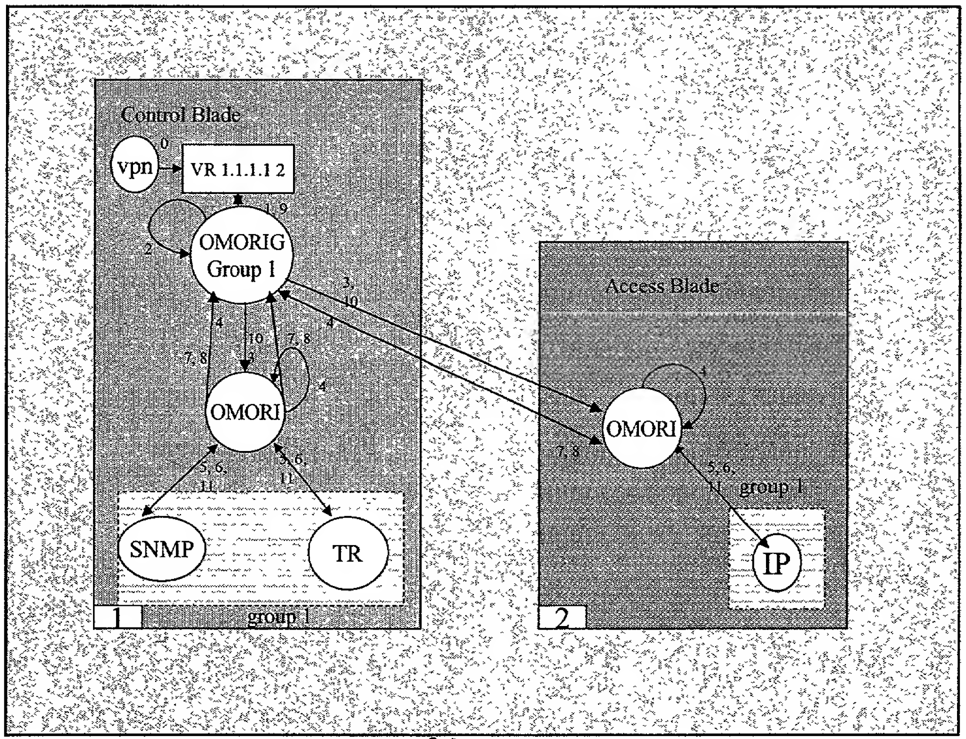


Figure 22. VR Deletion

Step	OMCD	OMORIG	OMORI	Object
0	For specified VPN and VR lookup vr_descriptor. Call OMORIG to delete corresponding group.			
1		Lookup group descriptor by specified id. Filter OMORIs which have objects to be destroyed(which belong to the specified group)		
2			Receive DESTROY_GROUP_ OBJECTS request for specified group. Lookup a group; Send generic IOCTL STOP_OBJECT to every local object, which belongs to the group	
3				Stop operating, destroy all connections with other objects

Fig. 23a

4			Send generic IOCTL DESTROY_OBJECT to every local object, which belongs to the group	
5				Free itself
6			If FAILURE remove object from the group, send MV_OBJ_TO_GROUP and RM_OBJ_FROM_GROUP to OMORIG;	
7		Receive MV_OBJ_TO_GROUP request, move object to the group 0(OM_BASE_GROUP); Receive RM_OBJ_FROM_GROUP request; remove object id from OMORIG Database		
8		Receive Object DESTROY_GROUP_OBJECTS. Subtract number of destroyed objects from the total number of objects in the group (VR). If all objects destroyed then signal to OMCD, otherwise do nothing.		
8	VR destroyed, return status to user			

Fig. 23b

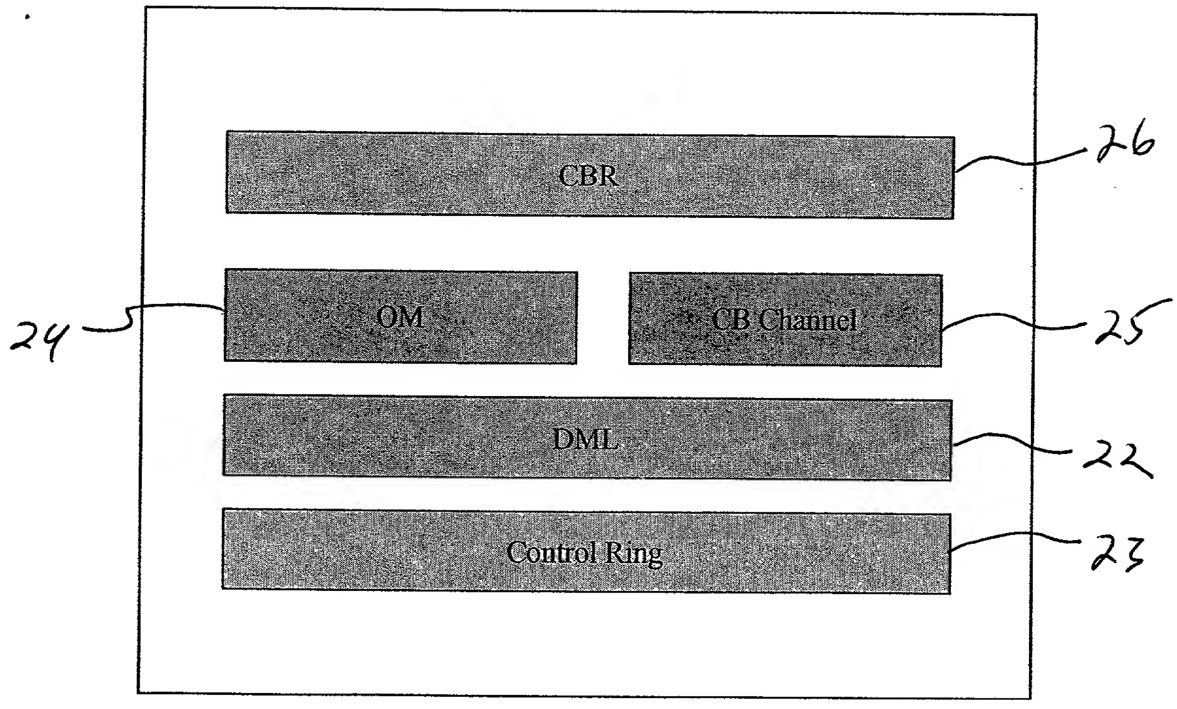


Fig. 24

~~2. CBR Design~~

In one embodiment Control Blade ~~redundancy~~ (CBR) 26 ~~to~~ is designed to create and maintain replicas of the Master Control Blade management information on the Standby Control Blades and to reuse ~~it in~~ that information the case of failure of the ~~current~~ Master and election of a ~~new~~ Master. The Control Ring normally elects the new Master. If the Control Ring ~~detection~~ mechanism fails a software-based leader election ~~protocol~~ implemented by DML will elect the new Master. This redundancy is illustrated in Figure 25.

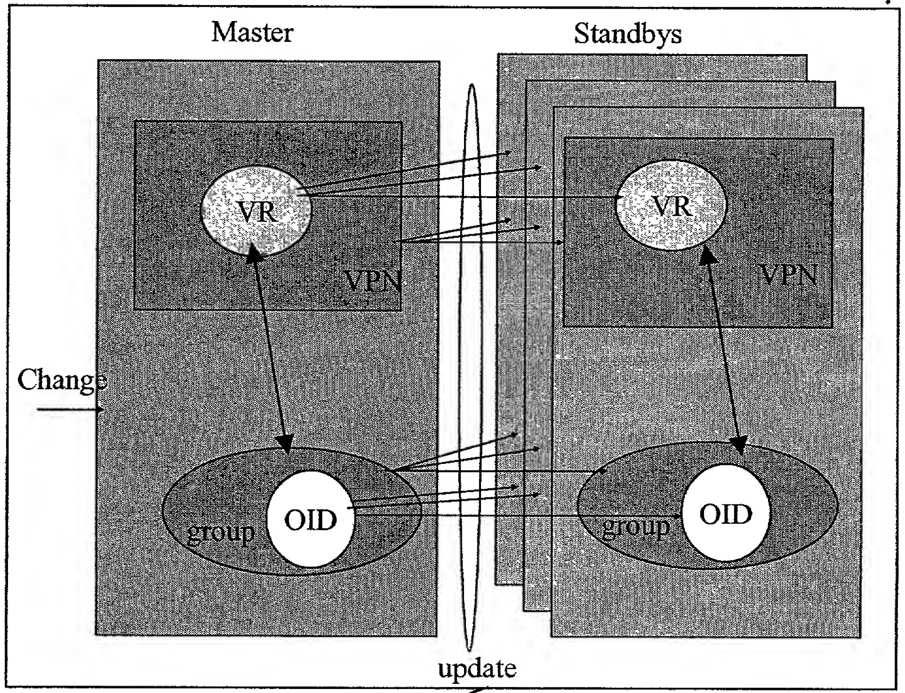


Figure ~~2~~ 25. Control Blade Redundancy

initialized to READY.

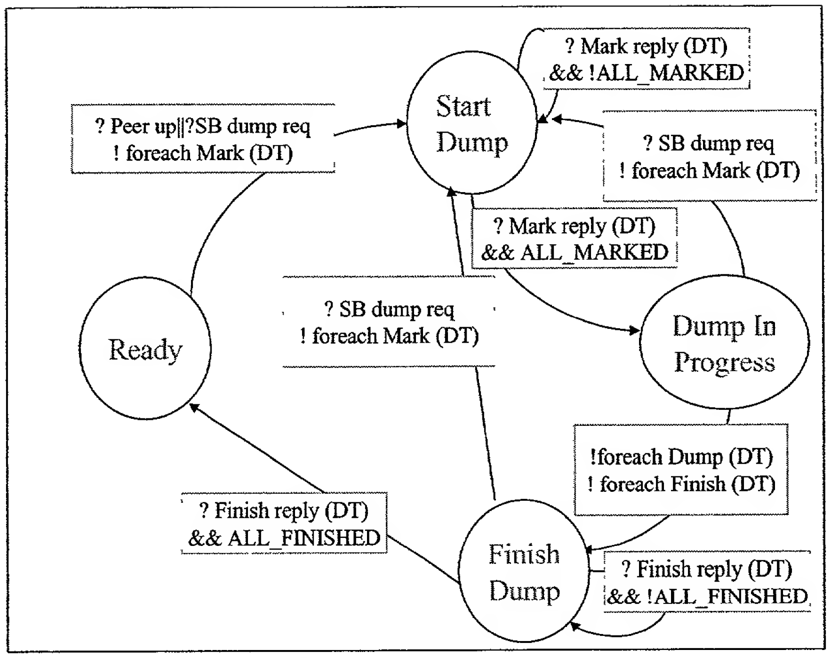
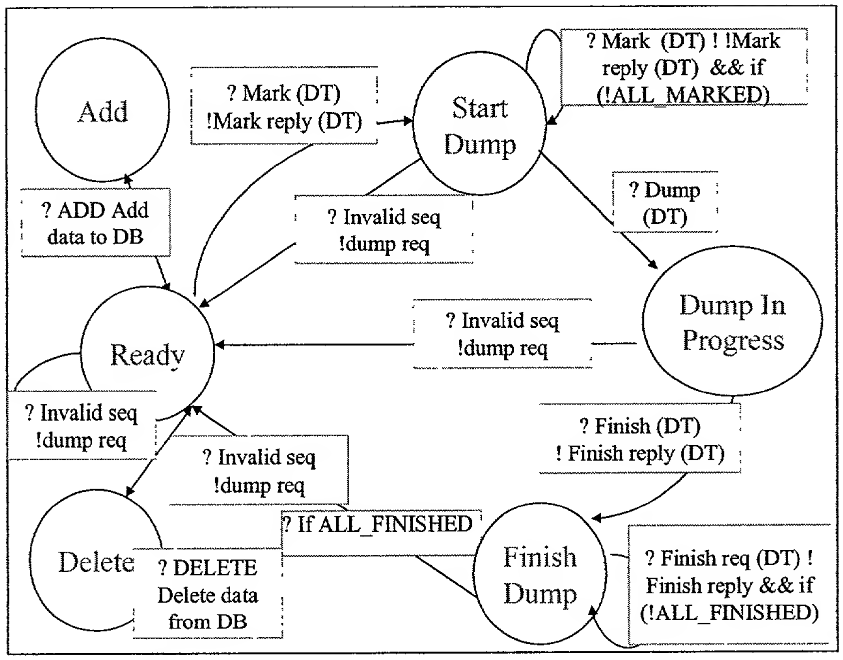

Start Dump
? Mark reply (DT) && !ALL_MARKED
? Peer up||?SB dump req ! foreach Mark (DT)
? SB dump req ! foreach Mark (DT)
? Mark reply (DT) && ALL_MARKED
Ready
? SB dump req ! foreach Mark (DT)
Dump In Progress
!foreach Dump (DT) ! foreach Finish (DT)
? Finish reply (DT) && ALL_FINISHED
Finish Dump
? Finish reply (DT) && !ALL_FINISHED

Figure ~~3~~26. CBR DUMP Master State Transition Diagram



Add
? Mark (DT) !Mark reply (DT)
Start Dump
? Mark (DT) ! !Mark reply (DT) && if (!ALL_MARKED)
? ADD Add data to DB
? Invalid seq !dump req
? Dump (DT)
Dump In Progress
Ready
? Invalid seq !dump req
? Invalid seq !dump req
? Finish (DT) ! Finish reply (DT)
? Invalid seq !dump req
? If ALL_FINISHED
Finish Dump
Delete
? DELETE Delete data from DB
? Finish req (DT) ! Finish reply && if (!ALL_FINISHED)

Figure ~~4~~ 27. CBR DUMP Standby State Transition Diagram

Attorney Docket No. 1384.006US1

SCHWEGMAN ■ LUNDBERG ■ WOESSNER ■ KLUTH

United States Patent Application

COMBINED DECLARATION AND POWER OF ATTORNEY

As a below named inventor I hereby declare that: my residence, post office address and citizenship are as stated below next to my name; that

I verily believe I am the original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled: **SWITCH MANAGEMENT SYSTEM AND METHOD**.

The specification of which is attached hereto.

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with 37 C.F.R. § 1.56 (attached hereto). I also acknowledge my duty to disclose all information known to be material to patentability which became available between a filing date of a prior application and the national or PCT international filing date in the event this is a Continuation-In-Part application in accordance with 37 C.F.R. §1.63(e).

I hereby claim foreign priority benefits under 35 U.S.C. §119(a)-(d) or 365(b) of any foreign application(s) for patent or inventor's certificate, or 365(a) of any PCT international application which designated at least one country other than the United States of America, listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on the basis of which priority is claimed:

No such claim for priority is being made at this time.

I hereby claim the benefit under 35 U.S.C. § 119(e) of any United States provisional application(s) listed below:

No such claim for priority is being made at this time.

I hereby claim the benefit under 35 U.S.C. § 120 or 365(c) of any United States and PCT international application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT international application in the manner provided by the first paragraph of 35 U.S.C. § 112, I acknowledge the duty to disclose material information as defined in 37 C.F.R. § 1.56(a) which became available between the filing date of the prior application and the national or PCT international filing date of this application:

No such claim for priority is being made at this time.

I hereby appoint the following attorney(s) and/or patent agent(s) to prosecute this application and to transact all business in the Patent and Trademark Office connected herewith:

Anglin, J. Michael	Reg. No. 24,916	Huebsch, Joseph C.	Reg. No. 42,673	Nelson, Albin J.	Reg. No. 28,650
Bianchi, Timothy E.	Reg. No. 39,610	Jurkovich, Patti J.	Reg. No. 44,813	Nielsen, Walter W.	Reg. No. 25,539
Billion, Richard E.	Reg. No. 32,836	Kalis, Janal M.	Reg. No. 37,650	Oh, Allen J.	Reg. No. 42,047
Black, David W.	Reg. No. 42,331	Kaufmann, John D.	Reg. No. 24,017	Padys, Danny J.	Reg. No. 35,635
Brennan, Leoniede M.	Reg. No. 35,832	Klima-Silberg, Catherine I.	Reg. No. 40,052	Parker, J. Kevin	Reg. No. 33,024
Brennan, Thomas F.	Reg. No. 35,075	Kluth, Daniel J.	Reg. No. 32,146	Perdok, Monique M.	Reg. No. 42,989
Brooks, Edward J., III	Reg. No. 40,925	Lacy, Rodney L.	Reg. No. 41,136	Prout, William F.	Reg. No. 33,995
Chu, Dinh C.P.	Reg. No. 41,676	Lemaire, Charles A.	Reg. No. 36,198	Schumm, Sherry W.	Reg. No. 39,422
Clark, Barbara J.	Reg. No. 38,107	LeMoine, Dana B.	Reg. No. 40,062	Schwegman, Micheal L.	Reg. No. 25,816
Clise, Timothy B.	Reg. No. 40,957	Lundberg, Steven W.	Reg. No. 30,568	Scott, John C.	Reg. No. 38,613
Dahl, John M.	Reg. No. 44,639	Maeyaert, Paul L.	Reg. No. 40,076	Smith, Michael G.	Reg. No. 45,368
Drake, Eduardo E.	Reg. No. 40,594	Maki, Peter C.	Reg. No. 42,832	Speier, Gary J.	Reg. No. 45,458
Embretson, Janet E.	Reg. No. 39,665	Malen, Peter L.	Reg. No. 44,894	Steffey, Charles E.	Reg. No. 25,179
Fordenbacher, Paul J.	Reg. No. 42,546	Mates, Robert E.	Reg. No. 35,271	Terry, Kathleen R.	Reg. No. 31,884
Forrest, Bradley A.	Reg. No. 30,837	McCrackin, Ann M.	Reg. No. 42,858	Tong, Viet V.	Reg. No. 45,416
Gamon, Owen J.	Reg. No. 36,143	Moore, Charles L., Jr.	Reg. No. 33,742	Viksnins, Ann S.	Reg. No. 37,748
Harris, Robert J.	Reg. No. 37,346	Nama, Kash	Reg. No. 44,255	Woessner, Warren D.	Reg. No. 30,440

I hereby authorize them to act and rely on instructions from and communicate directly with the person/assignee/attorney/firm/organization/who/which first sends/sent this case to them and by whom/which I hereby declare that I have consented after full disclosure to be represented unless/until I instruct Schwegman, Lundberg, Woessner & Kluth, P.A. to the contrary.

Please direct all correspondence in this case to **Schwegman, Lundberg, Woessner & Kluth, P.A.** at the address indicated below:

P.O. Box 2938, Minneapolis, MN 55402
Telephone No. (612)373-6900

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of joint inventor number 1 : **Abraham R. Matthews**
Citizenship: **India** Residence: **San Jose, CA**
Post Office Address: 933 Willowleaf Drive
Apt. 605
San Jose, CA 95128

Signature: ______________________ Date: ______________________
Abraham R. Matthews

Full Name of joint inventor number 2 : **Anna Berenberg**
Citizenship: **United States of America** Residence: **Redwood City, CA**
Post Office Address: 3200 Bridge Parkway
Redwood City, CA 94065

Signature: ______________________ Date: ______________________
Anna Berenberg

§ 1.56 Duty to disclose information material to patentability.

(a) A patent by its very nature is affected with a public interest. The public interest is best served, and the most effective patent examination occurs when, at the time an application is being examined, the Office is aware of and evaluates the teachings of all information material to patentability. Each individual associated with the filing and prosecution of a patent application has a duty of candor and good faith in dealing with the Office, which includes a duty to disclose to the Office all information known to that individual to be material to patentability as defined in this section. The duty to disclose information exists with respect to each pending claim until the claim is canceled or withdrawn from consideration, or the application becomes abandoned. Information material to the patentability of a claim that is canceled or withdrawn from consideration need not be submitted if the information is not material to the patentability of any claim remaining under consideration in the application. There is no duty to submit information which is not material to the patentability of any existing claim. The duty to disclose all information known to be material to patentability is deemed to be satisfied if all information known to be material to patentability of any claim issued in a patent was cited by the Office or submitted to the Office in the manner prescribed by §§ 1.97(b)-(d) and 1.98. However, no patent will be granted on an application in connection with which fraud on the Office was practiced or attempted or the duty of disclosure was violated through bad faith or intentional misconduct. The Office encourages applicants to carefully examine:

(1) prior art cited in search reports of a foreign patent office in a counterpart application, and

(2) the closest information over which individuals associated with the filing or prosecution of a patent application believe any pending claim patentably defines, to make sure that any material information contained therein is disclosed to the Office.

(b) Under this section, information is material to patentability when it is not cumulative to information already of record or being made of record in the application, and

(1) It establishes, by itself or in combination with other information, a prima facie case of unpatentability of a claim; or

(2) It refutes, or is inconsistent with, a position the applicant takes in:

(i) Opposing an argument of unpatentability relied on by the Office, or

(ii) Asserting an argument of patentability.

A prima facie case of unpatentability is established when the information compels a conclusion that a claim is unpatentable under the preponderance of evidence, burden-of-proof standard, giving each term in the claim its broadest reasonable construction consistent with the specification, and before any consideration is given to evidence which may be submitted in an attempt to establish a contrary conclusion of patentability.

(c) Individuals associated with the filing or prosecution of a patent application within the meaning of this section are:

(1) Each inventor named in the application:

(2) Each attorney or agent who prepares or prosecutes the application; and

(3) Every other person who is substantively involved in the preparation or prosecution of the application and who is associated with the inventor, with the assignee or with anyone to whom there is an obligation to assign the application.

(d) Individuals other than the attorney, agent or inventor may comply with this section by disclosing information to the attorney, agent, or inventor.

Appendix A

7. OM Implementation Details

7.1. OMCD

7.1.1. Data Structures

```
typedef
struct vpn_descriptor_s
{
   TAG_DECL;                             /* run time type identification */
   dlcl_list_t       list;               /* List of all VPNs */
   dlcl_list_t       vr_list;            /* list of all VR for this VPN */
   lck_mutex_t       vr_lock;
   uint32_t       id;                    /* Globally unique VPN id */
   int               numb_of_vrs;           /* Number of VR's in this VPN */
   char           name[NAMELEN+1];       /* VPN name */
   int               admin_status;          /* Administrative status of the
VPN */
#define VPN_ADM_CREATE     0x0001            /* event */
#define VPN_ADM_GO         0x0002            /* event */
#define VPN_ADM_SUSPEND    0x0004            /* event */
#define VPN_ADM_DESTROY    0x0008            /* event */
#define VPN_ADM_NOT_EXIST  0x0100            /* state */
#define VPN_ADM_INACTIVE   0x0200            /* state */
#define VPN_ADM_ACTIVE     0x0400            /* state */
   int               oper_status;        /* Operational status of the VPN */
#define VPN_OPER_DOWN         0x01
#define VPN_OPER_INIT         0x02
#define VPN_OPER_UP           0x04
#define VPN_OPER_FAULT_RECOVERY  0x08
   int               marked;       /* Used to delete vpn in CBR data base */
   long           timestamp;       /*  Time when data was send to CBR data
base */
}  vpn_descriptor_t;

typedef
struct vr_descriptor_s
{
   TAG_DECL;                             /* run time type identification */
   dlcl_list_t       list;               /* List of all VRs in a VPN */
   ipaddr_t       id;                    /* Identifies VR */
   vpn_descriptor_t  *vpn;               /*  Identifies  VPN  to  which  VR
belongs */
   object_group_id_t group;      /* Identifies group to which VR mapped */
   int            admin_status;  /* Administrative status of the VR */
#define VR_ADM_CREATE      0x0001            /* event */
#define VR_ADM_GO       0x0002         /* event */
#define VR_ADM_SUSPEND     0x0004            /* event */
#define VR_ADM_DESTROY     0x0008         /* event */
#define VR_ADM_NOT_EXIST   0x0100         /* state */
#define VR_ADM_INACTIVE    0x0200         /* state */
#define VR_ADM_ACTIVE      0x0400         /* state */
   int            oper_status;   /* Operational status of the VR */
#define VR_OPER_DOWN       0x01
#define VR_OPER_INIT       0x02
#define VR_OPER_UP            0x04
#define VR_OPER_FAULT_RECOVERY   0x08
   int            marked;  /* Used to delete vpn in CBR data base */
   long           timestamp;       /*  Time  when  data  was  send  to  CBR
data base */
}  vr_descriptor_t;
```

7.1.2. API

7.1.2.1. Create / Destroy API

FUNCTION	EFFECT	ARGUMENTS
`om_create_vpn`	Create VPN with specified VPN ID. Returns SUCCESS or FAILURE.	VPN ID
`om_create_vr`	Create VR for existing VPN with specified VR ID. Returns SUCCESS or FAILURE.	VPN ID, VR ID
`om_destroy_vpn`	Destroys VPN and all VRs configured in the VPN. Returns SUCCESS or FAILURE.	VPN ID
`om_destroy_vr`	Destroys VR. Returns SUCCESS or FAILURE.	VPN ID, VR ID

7.1.2.2. IOCTL API

FUNCTION	EFFECT	ARGUMENTS
`om_ioctl`	Sends IOCTL with data to object of the specified class, which belongs to the VR, identified by < VPN ID, VR ID>. Returns SUCCESS or FAILURE in case of system failure or any other status supplied by the object-specific IOCTL handler.	VPN ID, VR ID, class id, IOCTL code, data pointer, length
`om_ioctl_by_obj_id`	Sends IOCTL with data to object identified by supplied object id. Returns SUCCESS or FAILURE in case of system failure or any other status supplied by the object-specific IOCTL handler.	Object ID IOCTL code, data pointer, length

7.1.2.3. Check existence

FUNCTION	EFFECT	ARGUMENTS
`om_is_vpn_exist`	Checks the existence of the VR, which is specified by the <VPN ID, VR ID>. Return TRUE if such a VPN exist and FALSE otherwise	VPN ID
`om_is_vr_exist`	Checks the existence of the VPN, which is specified by the VPN ID. Return TRUE if such a VPN exist and FALSE otherwise	VPN ID, VR ID

7.1.2.4. Attribute Access

FUNCTION	EFFECT	ARGUMENTS
`om_get_first_vpn`	Looks up first VPN ih the global list of VPNs in the single IPSX system. Returns ID of the first VPN if at least one VPN exists, and 0 otherwise.	
`om_get_next_vpn`	Searches the global list of the VPNs on the single Orion for the next VPN after the one, specified by VPN ID. If VPN ID is 0, then it returns VPN ID of the first VPN. If the VPN specified by VPN ID does not exist, then function returns next larger VPN ID than specified one. Returns 0 if no VPN was created, or there is no VPN with VPN ID and there is no next VPN.	VPN ID
`om_get_first_vr`	Looks up first VR in the list of the VRs, which belong to the VPN with VPN ID in the single Orion. Returns pointer to the VR ID at least one VR exists, and NULL otherwise. Returns NULL if there is no VPN with supplied VPN ID.	VPN ID
`om_get_next_vr`	Looks up the next VR after one, specified by the VR ID in the list of the VRs, which belong to the VPN with VPN ID in the single Orion. If VPN ID is not valid, then function returns NULL. If VR with specified VR ID does not exist, then function returns a pointer to the VR with next valid VR ID. Returns NULL if no VR was created in VPN with	VPN ID, VR ID

	VPN ID, or there is no next VR after VR with specified VR ID.	
om_get_vpn_name	Gets a name of a VPN, specified by the VPN ID. SUCCESS is returned if the VPN with VPN ID was found and FAILURE is returned otherwise.	VPN ID, name pointer, name size
om_set_vpn_name	Sets a name of a VPN, specified by the VPN ID. SUCCESS is returned if the VPN with VPN ID was found and FAILURE is returned otherwise.	VPN ID, name pointer, name size
om_get_vr_name	Gets a name of a VR, specified by the VPN ID and VR ID. SUCCESS is returned if the VR was found and FAILURE is returned otherwise.	VPN ID, VR ID, name pointer, name size
om_set_vr_name	Sets a name of a VR, specified by the <VPN ID, VR ID>. SUCCESS is returned if the VR was found and FAILURE is returned otherwise.	VPN ID, VR ID, name pointer, name size
om_get_vr_type	Gets a type of a VR, specified by the VPN ID and VR ID. SUCCESS is returned if the VR was found and FAILURE is returned otherwise.	VPN ID, VR ID, VR type
om_set_vr_type	Sets a type of a VR, specified by the <VPN ID, VR ID>. SUCCESS is returned if the VR was found and FAILURE is returned otherwise.	VPN ID, VR ID, VR type

7.2. OMORIG

7.2.1. Data Structures

```
typedef
struct oid_link_s
{
   TAG_DECL;
   dlcl_list_t       grp_lst;       /* List of all object IDs in the
group */
   dlcl_list_t       omori_lst;     /* List of all object IDs in the
group */
   dlcl_list_t       global_lst;    /* Sorted Global List of all object
IDs in the system */
   dlcl_list_t       list;          /* Global List of all object IDs in
the system */
   object_id_t       id;
   int              marked;        /* Used to delete oid in CBR data
base */
   long          timestamp;        /* Time when data was send to CBR
data base */
}  oid_link_t;
```

7.2.2. Message Tags

```
/* OMORI and OMORIG reply tag */
#define  OM_REPLY_TAG                0x4000
The following tags are sent by OMORIG to OMORI
#define  OMORI_DESTROY_GRP_OBJS_TAG 0x4003
#define  OMORI_MV_OBJ_TO_GRP_TAG    0x4005
#define  OMORI_CREATE_OBJ_TAG       0x4009
#define  OMORI_DESTROY_OBJ_TAG      0x4011
#define  OMORI_ACTIVATE_OBJ_TAG     0X4013
#define  OMORI_STOP_OBJ_TAG         0x4015
#define  OMORI_FORWARD_OBJ_ID_TAG   0x4019
#define  OMORI_IOCTL_TAG            0x4021
#define  OMORI_UPDATE_VR_ATTR_TAG   0x4023
#define  OMORI_UPDATE_GRP_TAG       0x4025
#define  OMORI_UPDATE_OBJ_TAG       0x4027
```

7.2.3. API

7.2.3.1. Group Membership Change

FUNCTION	EFFECT	ARGUMENTS
`omorig_add_obj_id`	Adds object ID to the OM Global database. Returns SUCCESS or FAILURE	Object ID
`omorig_remove_obj_id`	Removes object ID from the OM Global database. Returns SUCCESS or FAILURE	Object ID

7.2.3.2. Attribute Access

FUNCTION	EFFECT	ARGUMENTS
`omorig_get_first_group`	Returns pointer to the first group in the DB	Pointer to the group hash table
`omorig_get_next_group`	Returns pointer to the next after the specified group in the DB	Pointer to the group hash table and the pointer to the group
`omorig_lookup_group_by_id`	Looks up group by the specified group ID. Returns pointer to the group descriptor if found and	GROUP ID

	NULL otherwise.	
omorig_lookup_oid	Looks up OID link in the DB by specified ID. Returns pointer to the OID link if found and NULL otherwise.	Object ID

7.3. OMORI

7.3.1. Data Structures

```
typedef
struct obj_descriptor_s
{
   TAG_DECL;                          /* run time type identification */
   dlcl_list_t    global_list;        /* Global list of all objects */
   dlcl_list_t    list;         /* List of all objects in the group */
   object_id_t    id;   /* Unique object id in the group context */
   int            class_id;       /* Identifies class of the object */
   uint32_t       vpn_id;  /* Identifies VPN to which object belongs */
   ipaddr_t       vr_id;      /* Identifies VR */
   global_address_t  obj_instance;     /*   Identifies   the   object   in
different address space */
   uint32_t       timestamp;     /* Time when object was registered */
}  obj_descriptor_t;
```

7.3.2. Message Tags

The following tags are sent by OMORI to OMORIG

```
#define  OMORIG_ADD_OBJ_ID_TAG      0x5005
#define  OMORIG_RM_OBJ_ID_TAG       0x5006
#define  OMORIG_MV_OBJ_ID_TAG       0x5007
#define  OMORIG_FORWARD_OBJ_ID_TAG  0x5009
#define  OMORIG_REGISTER_FUNS_TAG   0x5011
#define  OMORIG_UPDATE_OBJ_ID_TAG   0x5013
#define  OMORIG_UPDATE_GRP_TAG      0x5014
#define  OMORI_FRWD_IOCTL_TAG       0x5015
#define  OMORI_ID_REQ_TAG           0x5017
#define  OMORI_CLASS_OID_REQ_TAG    0x5019
#define  OMORIG_UPDATE_GRP_DONE_TAG 0x5026
#define  OMORIG_UPDATE_OBJ_DONE_TAG 0x5028
```

7.3.3. API

7.3.3.1. Create / Destroy API

FUNCTION	EFFECT	ARGUMENTS

`omori_obj_register`	Registers object to the OM Database. Returns SUCCESS or FAILURE	CLASS ID, Object ID, pointer to object
`omori_obj_deregister`	Deregisters object from the OM Database. Returns SUCCESS or FAILURE	Object ID
`omori_assign_and_register_obj_id`	Registers object to the OM database, but requests OM to assign object ID to the registered object. Returns SUCCESS or FAILURE	CLASS ID, Object ID, pointer to object
`omori_create_obj`	Creates object of specified class in the specified group. If object of this class in the group already exists and unique flag is TRUE, then existing object ID will be returned, otherwise new object will be created.	VPN ID, VR ID, GROUP ID, CLASS ID, address space id, unique flag. Returns object id.
`omori_create_active_obj`	Same as omori_create_obj but after successful creation sends ACTIVATE_OBJECT IOCTL to the object	VPN ID, VR ID, GROUP ID, CLASS ID, address space id, unique flag. Returns object id.
`omori_destroy_obj`	Sends STOP_OBJECT IOCTL to the object if it was in ACTIVE state and after this destroys object by sending DESTROY_OBJECT IOCTL to the object and then destroying all management information regarding this object.	Object ID

7.3.3.2.Group Membership Change

FUNCTION	EFFECT	ARGUMENTS
`omori_add_obj_to group`	Adds object to the specified group. If object reviously belong to another group it removes from the first group and add to the new. Returns SUCCESS or FAILURE	Object ID, GROUP ID
`omorig_remove_obj_id`	Removes object from the group where it belongs and moves it to the group 0 (OM_BASE_GROUP). Returns SUCCESS or FAILURE	Object ID

7.3.3.3.Attribute Access

FUNCTION	EFFECT	ARGUMENTS

omori_get_obj_group	Returns Group ID of the group to which object belongs, if object is not local then information is retrieved from the OMORIG	Object ID
omori_get_address_space	Returns address space ID of the group to which object belongs, if object is not local then information is retrieved from the OMORIG	Object ID
omori_get_local_obj_reference	Returns pointer to the local object and NULL if object is not local	Object ID

7.4. Transaction layer

7.4.1. Data Structures

```
typedef
struct om_request_s
{
   TAG_DECL;                                /* run time type identification */
   dlcl_list_t      list;                   /* List of requests */
   int              index;                  /* matching index */
   om_callback_f    *func;                  /* callback function */
   void             *cookie;
   int              tag;
   address_space_t  dest;                   /* Destination address */
}  om_request_t;
typedef
struct om_req_cookie_s
{
   TAG_DECL;                            /* run time type identification */
   int                 req_id;  /* Request ID */
   address_space_t     addr;    /* Address of the request originator */
   int                 tag;     /* message tag */
}  om_req_cookie_t;
```

7.4.2. API

FUNCTION	EFFECT	ARGUMENTS
om_send_request	Sends data to the destination, adds transaction control header in front of the packet, allocates a request, adds it to the request list. If callback function is NULL then free request.	Data packet, destination address, tag, callback function, request cookie.
om_recv_request	Receive request, allocate request cookie, save request id and source	Data packet

	address in it.	
om_send_reply	Sets message tag, depending on the channel, which data sent on. Sends data to the destination, adds transaction control header in front of the packet, free request cookie	Data packet, request cookie, tag, channels to send reply on.
om_recv_reply	Receive reply, calls callback function from the transaction control block (om_request_t). After completion free the request.	Data packet
om_get_request_tag	Retrieves message tag of the request.	Request cookie
om_get_request_src	Retrieves address of the originator of the request.	Request cookie
om_abort_requests_by_addr	Abort all the transactions associated with the address.	Address of the peer

Appendix B

3. CBR Data Structures

3.1. CBR Data Types

```
#define  CBR_VPN_DATA_TYPE        0
#define  CBR_VR_DATA_TYPE         1
#define  CBR_GROUP_DATA_TYPE      2
#define  CBR_OID_DATA_TYPE        3
```

3.2. CBR Message Tags

```
/* Control Blade Redundancy message tags */
#define  CBR_REPLY_TAG            0x6000
#define  CBR_MARK_TAG             0x6001
#define  CBR_DUMP_TAG             0x6003
#define  CBR_FINISH_TAG           0x6005
#define  CBR_ADD_TAG              0x6007
#define  CBR_DELETE_TAG           0x6009
#define  CBR_SB_DUMP_REQ_TAG      0x6011
#define  CBR_SEQ_TAG              0x6013
```

3.3. CBR Messages Format

```
typedef
struct   cbr_msg_s
{
   int                 type;        /* Identifies Data type */
   int                 seq;         /* Identifies sequence number */
   int                 status;      /* return status */
}  cbr_msg_t;
```

3.4. CBR Node Structure

```
typedef
struct cbr_node_s
{
   TAG_DECL;                        /* run time type identification */
   dlcl_list_t       list;          /* List of all potential masters in the
system */
   address_space_t      addr;          /* Address space of peer */
#define CBR_READY                 0
#define CBR_START_DUMP            1
#define CBR_DUMP_IN_PROGRESS      2
#define CBR_FINISH_DUMP           3
#define  CBR_UPDATE_ADD           4
#define  CBR_UPDATE_DELETE        5
   int                 state;          /* State of the peer */
} cbr_node_t;
```

4. CBR API

4.1. CBR Init

```
void cbr_init ();
```

This function

1. Initializes list of the CBR nodes, which will communicate in the CBR process and adds itself to the list.
2. Registers callout function to the DML to be called in case message for CB Channel is received.
3. Initializes CBR message sequence numbers.

This function has to be called in every CBR processing node.

4.2. CBR Peer Up/Down

```
void cbr_peer_up (IN address_space_t addr);
```

1. Runs only on Master Control Blade
2. Does not run if peer up notification came with local address id (itself)
3. Adds peer to the CBR Node list
4. Starts dump database for peer

```
void cbr_peer_down (IN  address_space_t addr);
```

1. Runs only on Master Control Blade
2. Does not run if peer down notification came with the local address id (itself)
3. Removes peer from the CBR Node list

4.3. Update

Actions: CBR_ADD_ACTION and CBR_DELETE_ACTION

```
void cbr_master_update_oid (
   IN oid_link_t        *oid,
   IN int               action);
```

Packages provided OID and sends packet to every Standby CB

```
void cbr_master_update_group (
   IN omorig_group_t    *grp,
   IN int               action);
```

Packages provided GROUP and sends packet to every Standby CB

```
void cbr_master_update_vr (
   IN vr_descriptor_t      *vrdp,
   IN int               action);
```

Packages provided VR and sends packet to every Standby CB

```
void cbr_master_update_vpn (
   IN vpn_descriptor_t  *vpndp,
   IN int               action);
```
Packages provided VPN and sends packet to every Standby CB

These functions have to be called only on Master CB by OMORIG.

4.4. OM API used by Standby nodes

Standby nodes after receiving messages with the specified action for OM Global Database change make changes using the following set of API provided by the OM. The detailed explanation of API provided to the application is given in [5]. The detailed explanation of API, which is common to OM and CBR, is provided in the [4].

```
IMPORT   omorig_group_t    *omorig_get_first_group (
   IN void           *set);
```
Returns a pointer to the first group in the DB.

```
IMPORT   omorig_group_t    *omorig_get_next_group (
   IN void           *set,
   IN void           *elem );
```
Returns a pointer to the next after the specified one group in the DB.

```
IMPORT   vr_descriptor_t      *omdb_create_vr (
   IN uint32_t       vpn_id,
   IN ipaddr_t       *vr_id   );
```
Creates VR descriptor for specified vpn id and vr id and fills in with default values.

```
IMPORT   omorig_group_t    *omdb_create_group (
   IN uint32_t       vpn_id,
   IN ipaddr_t       *vr_id,
   IN int               class_selector_flag  );
```
Creates Group descriptor for specified vpn id and vr id and fills in with default values. Class selector flag defines set of object classes, which are mandatory to be created for group to be created.

```
IMPORT   int omdb_destroy_vr (
   IN uint32_t    vpn_id,
   IN ipaddr_t    *vr_id   );
```
Destroys VR descriptor for specified vpn id and vr id.

```
IMPORT   int   omdb_delete_group (
   IN  void    *arg1,
   IN  void    *arg2 );
```
Destroys Group descriptor after all the object destoyed.

```
IMPORT omorig_group_t      *omorig_lookup_group_by_id (
   IN object_group_id_t group_id );
```
Looks up group by the specified group ID.

```
IMPORT   int   omorig_add_obj_id (
   IN object_id_t       *obj_id  );
```
Adds object ID to the OM Global Database.

```
IMPORT   int   omorig_remove_obj_id (
   IN void           *flag,
   IN void           *obj_id  );
```
Remove object ID from the OM Global Database.

```
IMPORT   oid_link_t  *omorig_lookup_oid (
   IN object_id_t       *id   );
```
Finds OID link in the OM Global Database by the specified object ID.

```
IMPORT int om_create_vpn  (
   IN uint32_t       vpn_id);
```
Creates VPN descriptor and fills in with default values.